Universidad Carlos III de Madrid
Escuela Politécnica Superior

Grado en Ingeniería Informática
Trabajo de Fin de Grado

Estudio de gestión de sistemas basados en grafos

Autor: Roberto Monsalve Toledo
Tutora: Anabel Fraga Vázquez
Cotutor: José María Álvarez Rodríguez

Agradecimientos

Después de mucho esfuerzo, sacrificio y constancia, muchas alegrías y algún contratiempo, me encuentro hoy aquí, escribiendo los agradecimientos de mi Trabajo de Fin de Grado, con el que cierro una bonita etapa universitaria y abro otra lleno de ilusión, esperanza y optimismo.

En primer lugar y antes que nada, me gustaría dar las gracias a dos personas fundamentales en mi vida y que gracias a ellos, he tenido la posibilidad de formarme en todo lo que he querido, incluyendo estos estudios universitarios. Esas dos personas son Julia y Luis, mis padres. Gracias por darme siempre vuestro apoyo en todas y cada una de las decisiones que he ido tomando en la vida. Muchas gracias por todo.

En el ámbito de la universidad, quisiera agradecer a mi tutora Anabel Fraga Vázquez, la oportunidad que me dio de realizar este proyecto. Asimismo, me gustaría agradecer a mi cotutor José María Álvarez Rodríguez, todo lo que me ha ayudado en la realización de este trabajo, por dedicar parte de su tiempo a resolver mis dudas.

A lo largo de esta etapa en la universidad he conocido a mucha gente y he tenido muy buenos compañeros, pero merecen mención especial tres personas, que no sólo han sido mis compañeros sino también grandes amigos que me llevo para siempre. Gracias a Alejandro Martín, con quién más trato tuve al comienzo de la carrera, por aguantarme en aquellas tardes que pasábamos en la uni haciendo prácticas. Los dos últimos años de carrera los he compartido, casi en su totalidad, con Víctor García y Ana Gómez. Gracias también a vosotros por los grandes momentos que hemos vivido juntos y por los que vendrán. Los tres habéis sido grandes compañeros, pero sois aún más grandes como personas.

Volviendo al ámbito personal, también me gustaría dar las gracias a mi tía Emi, que para mi es como mi segunda madre y que siempre ha estado ahí para ayudarme cuando lo he necesitado. Agradecer a Cecilia por compartir su vida conmigo, darme en todo momento su apoyo, mensajes de ánimo y por ser la persona más especial que he conocido nunca. Gracias también a mi hermano, que siempre me apoya cuando lo necesito.

Por último, quisiera acordarme de Rubén Gómez y Sergio Cendrero, amigos de toda la vida, con los que he compartido grandes momentos y también han estado ahí siempre que lo he necesitado.

Muchas gracias a todos vosotros por haber hecho posible todo esto.

Resumen

Desde hace unos años, tienen un auge cada vez más fuerte los sistemas de gestión de bases de datos NoSQL. Este incremento en el uso de estas herramientas viene dado por el constante aumento del número de datos que las organizaciones deben manejar a diario y por los nuevos desafíos que propone Internet. Estos sistemas ofrecen la oportunidad de manejar grandes volúmenes de datos (lo que se conoce como Big Data) y afrontan los desafíos que propone internet mejor que los gestores de bases de datos tradicionales.

Existen distintos tipos de bases de datos NoSQL, como las bases de datos basadas en documentos, en grafos, las que almacenan la información en columnas y las que almacenan clave-valor.

El presente Trabajo de Fin de Grado se enfoca en las bases de datos basadas en grafos. Se pretende realizar un análisis de los principales sistemas gestores de este tipo de bases de datos repasando las características más importantes de cada uno de ellos. Además, este trabajo tiene como objetivo realizar una comparativa entre el modelo RSHP y Neo4j, para comprobar cuál de los dos sistemas recupera mejor la información relevante a términos concretos.

Para todo ello, en primer lugar se analizan las principales características de sistemas como Neo4j, KnowledgeMANAGER, Sparksee, GraphBase, Infinite Graph, entre algunos otros. Se presta mayor atención a Neo4j y KnowledgeMANAGER, ya que son los dos sistemas utilizados para realizar la comparativa.

Después de analizar distintas bases de datos orientadas a grafos, se detallan diferentes formas de almacenar la información en el grafo de Neo4j, comparándolas entre sí y escogiendo la más adecuada para la comparativa posterior. Asimismo, otro aspecto importante de este trabajo es la creación de un proceso automatizado que sea capaz de generar consultas de forma aleatoria y de ejecutarlas de forma programática en Neo4j. A través de este proceso, se obtienen las consultas en lenguaje natural para ejecutar en RSHP y las consultas en Cypher junto con los resultados obtenidos de la ejecución en Neo4j.

Por último, después de ejecutar las consultas en ambos sistemas, se identifican los resultados relevantes para cada una de ellas, comprobando cuál de los dos recupera mejor datos relevantes. Para analizar esos resultados, se calculan las medidas de precisión y recall para cada consulta.

Palabras clave: Neo4j, RSHP, KnowledgeMANAGER, bases de datos, grafos.

Abstract

Some years ago, new types of databases called NoSQL began to appear. In recent years, these databases are increasing due to the high volume of data that organizations must handle everyday. These databases offer the opportunity to handle large volumes of data (this is known as Big Data), facing Internet's challenges better than relational databases did before.

There are different types of NoSQL databases, such as databases based on documents, graphs, databases which store information in columns and others that store information in key-value.

This final degree project is focused on databases based on graphs. It aims to make an analysis of the main management systems for this kind of databases reviewing the most important features of each one. In addition, this work aims to make a comparison between the RSHP model and Neo4j, to see which of the two systems recovers the relevant information, on specific terms, better.

In order to perform this work, firstly, the main features of systems such as Neo4j, KnowledgeMANAGER, Sparksee, GraphBase, Infinite Graph, among others, are analysed. Neo4j and KnowledgeMANAGER are studied carefully because they are used to perform the comparison.

After analysing different databases based on graphs, different ways of storing information in the Neo4j graph are compared. Then, the most suitable alternative is chosen for subsequent comparative analysis. Also, another important point of this project is to create an automated process that is able to generate random queries and run them in Neo4j. Through this process, queries in Cypher, execution results in Neo4j and queries in natural language to execute in RSHP, are obtained.

Finally, after executing queries in both systems, relevant results for every query are identified. Then, it is checked whether a system gets better relevant data than the other one. To analyse these results, measures of precision and recall are calculated for each query.

Keywords: Neo4j, RSHP, KnowledgeMANAGER, databases, graphs.

ÍNDICE GENERAL

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

1. INTRODUCCIÓN

En este primer capítulo, se incluye una breve introducción en la que se plantea la problemática encontrada, se expone la motivación para llevar a cabo el trabajo, se indican los objetivos que se pretenden alcanzar a la finalización del mismo, se incluye un breve glosario de términos y se detalla la estructura del documento, incluyendo sobre qué trata cada uno de los capítulos.

1.1 PLANTEAMIENTO DEL PROBLEMA

Las bases de datos relacionales han sido durante mucho tiempo las más utilizadas por las empresas para almacenar y gestionar sus datos. Pero en los últimos tiempos, las tecnologías de la información han evolucionado de forma continua y rápida, lo que ha llevado a las organizaciones a adaptarse y a enfrentar nuevos desafíos. Esta evolución ha sido provocada, en parte, por un aumento considerable del volumen de datos que se manejan cada día.

Con estos nuevos retos surge el concepto de Big Data, el cual se refiere a ese gran conjunto de datos complejos que resulta difícil de procesar por los sistemas tradicionales. Principalmente, Big Data se caracteriza por sus conocidas "Tres Vs", variedad (diferentes tipos de datos, estructurados y no estructurados), volumen (alta escalabilidad, miles de nodos) y velocidad (los datos se procesan en los servidores donde se encuentran). Junto a este concepto, también surgen nuevos sistemas de gestión de bases de datos denominados NoSQL, como alternativa a los sistemas clásicos de gestión de bases de datos relacionales. Estos nuevos sistemas NoSQL se caracterizan, como bien indica su nombre, por no usar SQL como el principal lenguaje de consulta (algunos sistemas NoSQL sí lo soportan), además de no requerir estructuras fijas, como las tablas, para almacenar los datos.

Generalmente, las bases de datos NoSQL utilizan uno de los siguientes tres tipos de modelo de datos: modelo basado en documentos, modelo basado en grafos y modelo de clave-valor. El presente Trabajo de Fin de Grado nace con un afán de investigación enfocándose en las bases de datos basadas en grafos. Además de un análisis de estas bases de datos y de repasar las características de los sistemas de gestión más utilizados, en este trabajo se ha realizado una comparativa entre el modelo RSHP (utilizando la herramienta KnowledgeMANAGER) y Neo4j, para conocer cuál de los dos sistemas extrae mejor la información para términos concretos.

Se ha seleccionado Neo4j para este trabajo de investigación ya que, es uno de los sistemas de almacenamiento basados en grafos más utilizados hoy en día. Lufthansa Systems, Tomtom, Ebay, InfoJobs, LinkedIn o National Geographic, son algunos de sus clientes más importantes. Por su parte, se ha utilizado la herramienta KnowledgeMANAGER para este experimento ya que se pretendía incluir en la comparación un sistema relacionado con la gestión del conocimiento, pero que también se basase en grafos. Dicha herramienta se basa en el modelo RSHP, el cual crea un grafo implícito para almacenar la información y permite reutilizar el conocimiento. Por lo que el modelo RSHP, utilizando la citada herramienta, se convertía en uno de los elementos a incluir en este trabajo.

1.2 MOTIVACIÓN DEL TRABAJO

La motivación de este estudio surge del ímpetu de indagar e investigar el mundo de las bases de datos basadas en grafos. Este tipo de bases de datos, cada vez tienen más auge en el ámbito profesional. Esto supone un añadido a ese afán de investigación, ya que este proyecto ayuda a comprender en qué punto se encuentran hoy en día estas bases de datos y cómo de importantes pueden llegar a ser para las empresas. Este trabajo de investigación ofrece la oportunidad de adentrarse en el mundo de los grafos y conocer qué alternativas existen hoy en día en el mercado.

1.3 OBJETIVOS

En el presente Trabajo de Fin de Grado se han definido una serie de objetivos que deben ser cumplidos a la finalización del mismo. Estos objetivos se detallan a continuación:

- Analizar las diferentes alternativas, que existen en el mercado, de sistemas de gestión de bases de datos basadas en grafos.

- Realizar una comparativa, en lo que se refiere a extracción de información, entre Neo4j y el modelo RSHP (este último a través de la herramienta KnowledgeMANAGER). Dadas una serie de consultas con términos concretos, se pretende conocer cuál de los dos sistemas recupera mejor la información relevante a cada una de ellas. Se requiere el cálculo de la precisión y recall para cada consulta para comprobar cómo de buena es la recuperación.

- Analizar diferentes diseños a la hora de almacenar la información en el grafo en Neo4j, para seleccionar el que mejor se adapte para la comparativa realizada en el presente trabajo.

- Implementar un proceso de generación y ejecución automática de consultas en Neo4j. Las consultas se generarán, a partir del grafo creado en Neo4j, tanto en lenguaje Cypher, para Neo4j, como en lenguaje natural, para RSHP. Las consultas en lenguaje natural se ejecutarán en la herramienta KnowledgeMANAGER, mientras que las correspondiente a Neo4j, se ejecutarán programáticamente.

1.4 GLOSARIO DE TÉRMINOS

En este apartado se incluyen aquellos términos que son citados en el texto y no ha quedado claro cuál es su significado.

- **Ontología**: formulación de un exhaustivo y riguroso esquema conceptual dentro de uno o varios dominios dados, con la finalidad de facilitar la comunicación y el intercambio de información entre diferentes sistemas y entidades.

- **Tesauro**: es una lista de palabras o términos controlados, empleados para representar conceptos.

- **Benchmark**: técnica utilizada para medir el rendimiento de un sistema o componente del mismo. Frecuentemente, se utiliza para comparar pruebas de rendimiento sobre varios sistemas.

- **Framework**: en desarrollo de software, es una estructura conceptual y tecnológica de soporte definido, normalmente con módulos de software concretos, que puede servir de base para la organización y desarrollo de software.

1.5 ESTRUCTURA DEL DOCUMENTO

En este apartado se indican los diferentes capítulos que consta este documento, señalando brevemente de qué trata cada uno de ellos:

- **Introducción:** este apartado incluye el planteamiento del problema, junto con la motivación del trabajo, los objetivos definidos para el mismo y un glosario de términos cuyo significado no ha quedado claro en el texto.

- **Estado del arte:** este apartado incluye cuatro grandes secciones referidas a modelos de recuperación de información, lenguajes de recuperación de la información, sistemas de almacenamiento basados en grafos y benchmarks existentes entre algunos de esos sistemas.

- **Análisis de la solución:** en este capítulo se incluye el análisis realizado para elegir la mejor forma de almacenar los datos en el grafo de Neo4j. Asimismo, se incluyen apartados para la generación y ejecución automática de consultas.

- **Diseño del grafo en Neo4j:** en este capítulo se detalla el diseño final del grafo de Neo4j adoptado para el experimento realizado en el capítulo de Experimentación. Además, se incluyen distintas alternativas de diseño que se han pensado durante el trabajo. Cada decisión se justifica debidamente.

- **Implementación:** en este apartado se detalla la implementación llevada a cabo tanto para la creación e indexación del grafo en Neo4j, como para el proceso de generación y ejecución automática de consultas en Neo4j.

- **Experimentación:** en este capítulo se detalla el estudio realizado entre Neo4j y el modelo RSHP. Se explica en qué consiste el experimento, presentando los resultados obtenidos y analizándolos a través de distintas gráficas.

- **Gestión del proyecto:** en este apartado se incluye la planificación del presente Trabajo de Fin de Grado y el correspondiente presupuesto asociado al mismo.

- **Conclusiones del trabajo realizado y trabajos futuros:** se incluyen las conclusiones extraídas de la realización del trabajo y se indican las principales líneas a seguir para continuar con la investigación sobre el tema expuesto.

- **Bibliografía:** en este apartado se incluyen todas las referencias bibliográficas, las cuales se han citado, debidamente, a lo largo del documento. Cabe destacar que se ha utilizado el estándar IEEE.

- **Anexo A:** este anexo incluye un resumen de 10 páginas, escritas en inglés, del presente Trabajo de Fin de Grado.

- **Anexo B:** breve manual SQL.

2. ESTADO DEL ARTE

En este capítulo se realiza una introducción a las bases de datos orientadas a grafos (BDOG), incluyendo un análisis de los principales sistemas existentes hoy en día en el mercado. Asimismo, se detallan los principales modelos clásicos de recuperación de la información, los lenguajes de consulta más destacados y se citan algunos benchmarks existentes entre distintos sistemas de almacenamiento basados en grafos.

2.1 MODELOS DE RECUPERACIÓN DE INFORMACIÓN

En esta sección se van a repasar los principales modelos de recuperación de información existentes hoy en día. Concretamente, se van a ver los tres modelos clásicos: **booleano, vectorial y probabilístico**.

Los modelos de recuperación de información permiten recuperar información relevante para el usuario, de una colección de documentos dada. Los buscadores web (Google, Bing, Yahoo!), tan utilizados en los últimos tiempos, se basan en estos modelos para recuperar las entradas más relevantes para las preguntas (consultas en lenguaje natural) de los usuarios. Se componen básicamente de cuatro elementos representados en una cuádrupla del tipo **[D,Q,F,R(q_i, d_j)]** donde [1]:

- **D** son las representaciones de los documentos de la colección que se desea recuperar.
- **Q** son las representaciones de las consultas que reflejan las necesidades de información del usuario.
- **F** es un marco que permite establecer una relación entre las representaciones de los documentos y las de las consultas.
- **R(q_i, d_j)** o **Sim(q_i, d_j)** es la función de relevancia o similitud que asigna un peso al documento j para una consulta i dada.

En los modelos clásicos, los documentos son descritos por un conjunto de términos K={k_1…k_n}. Cada término tiene asignado un peso que puede variar según el documento que describa. La representación del peso del término 1 en el documento j sería: W_{1j}. Por lo que un documento podría describirse por los pesos de los términos que lo representan: d_j = {W_{1j} … W_{nj}}.

A continuación se procede a detallar las particularidades de cada uno de los tres modelos citados anteriormente.

2.1.1 MODELO BOOLEANO

El **modelo booleano** es un modelo simple, basado en la lógica de proposiciones y el álgebra booleana. Asigna pesos binarios teniendo en cuento si el término está o no en el documento. No existe relevancia parcial, son relevantes o no. Las consultas se construyen en lenguaje booleano a través de los operadores booleanos, como por ejemplo, And (+, ^, Y), Or (o, |, v), not (no, and not, -, ¬). Seguidamente se incluye un ejemplo con el que se entiende mejor el funcionamiento de este modelo de recuperación de información.

Documento 1: "…las películas se ruedan en escenas donde aparecen actores y actrices…"
Documento 2: "…en las películas actúan varios actores y actrices con distintos roles…"

Conjunto de términos K = {películas, escenas, actores, actrices, roles}
D1 = (1,1,1,1,0)
D2 = (1,0,1,1,1)

Consulta Q1: escenas AND (roles OR actores) = D1
Consulta Q2: actrices AND roles = D2

Figura 1: Ejemplo de consultas en modelo booleano

Como se puede observar en el ejemplo de la *Figura 1*, para cada documento, se asignan pesos binarios (1 si el término sí está en el documento y 0 en el caso de que no) a los términos elegidos en *K* (sobre los que se harán las consultas). Además, se incluyen dos ejemplos de consultas en las que se utilizan algunos de los operadores booleanos comentados anteriormente.

2.1.1.1 VENTAJAS DEL MODELO BOOLEANO

Algunas de las principales ventajas del modelo booleano son las siguientes:

- Presente en sistemas de bases de datos relacionales.
- Eficiente y simple.
- Utiliza álgebra booleana.
- Es útil para realizar diferentes experimentos.

2.1.1.2 DESVENTAJAS DEL MODELO BOOLEANO

En cuanto a las desventajas de este modelo, cabe destacar los siguientes puntos:

- No permite ordenación de los documentos por relevancia. Es decir, si el término está en el documento, se devuelve el documento sin importar el número de veces que aparezca dicho término.
- No tiene en cuenta la frecuencia del término.
- Con todos los operadores OR se obtienen demasiados resultados, mientras que con todos AND se recuperan muy pocos.

2.1.2 MODELO VECTORIAL

En el **modelo vectorial**, cada documento y consulta son representados por un vector, con tantas dimensiones como términos en K. Como se verá posteriormente con un ejemplo, se asignan pesos positivos y no binarios a los términos. La similitud entre una consulta y un documento se mide por la distancia que existe entre sus respectivos vectores como se puede apreciar en la *Figura 2*.

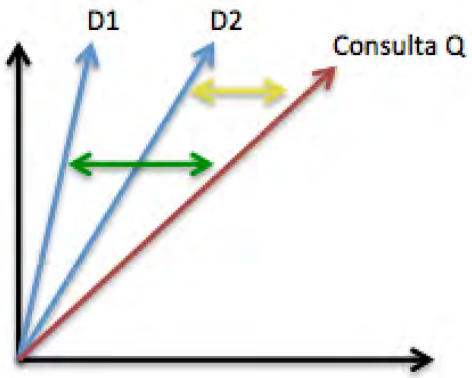


Figura 2: Representación de documentos y consultas en modelo vectorial

Se pueden utilizar diferentes funciones para realizar el cálculo del coeficiente de similitud, entre las que destacan: producto escalar, coseno del ángulo entre los dos vectores (la más utilizada), coeficiente de Dice y coeficiente de Jaccard. A continuación se van a detallar las dos primeras funciones.

2.1.2.1 SIMILITUD MEDIANTE EL PRODUCTO ESCALAR

La similitud mediante el producto escalar se calcula a través de la siguiente fórmula:

$$sim(d_j, q) = d_j \bullet q = \sum_{i=1}^{t} w_{ij} \cdot w_{iq}$$

Figura 3: Similitud mediante producto escalar

A partir de la *Figura 3*, se comprueba que la similitud mediante el producto escalar se calcula a través del sumatorio de los productos de W_{ij} (peso del término i en el documento j) por W_{iq} (peso del término i en la consulta q). Tomando el ejemplo del modelo booleano, a continuación se aplica al modelo vectorial con el producto escalar.

Documento 1: "…las películas se ruedan en escenas donde aparecen actores y actrices…"
Documento 2: "…en las películas actúan varios actores y actrices con distintos roles…"

Conjunto de términos K = {películas, escenas, actores, actrices, roles}
D1 = (5,3,2,2,0)
D2 = (3,0,1,1,4)
Q = (0,0,1,1,2)

Sim(D1, Q) = 2 • 1 + 2 • 1 = 4
Sim(D2, Q) = 1 • 1 + 1 • 1 + 4 • 2 = 10

Figura 4: Ejemplo de similitud mediante producto escalar

Como se puede observar en la *Figura 4*, se asignan diferentes pesos a los términos en función de su importancia en cada documento y también se asignan pesos a los términos en la consulta. Después, simplemente se realiza el producto escalar de cada documento con la consulta para determinar la similitud de cada uno de ellos con dicha consulta. En este caso,

según los pesos asignados de forma aleatoria, el documento 2 sería mucho más relevante para la consulta Q que el documento 1.

2.1.2.2 SIMILITUD MEDIANTE EL COSENO DEL ÁNGULO ENTRE LOS DOS VECTORES

La similitud mediante el coseno consiste en el producto escalar de ambos vectores normalizado por la longitud de los mismos. La fórmula correspondiente es la que se muestra a continuación:

$$CosSim(d_j, q) = \frac{\vec{d_j} \cdot \vec{q}}{\|\vec{d_j}\| \cdot \|\vec{q}\|} = \frac{\sum_{i=1}^{t} w_{ij} \cdot w_{iq}}{\sqrt{\sum_{i=1}^{t} w_{ij}^2} \cdot \sqrt{\sum_{i=1}^{t} w_{iq}^2}}$$

Figura 5: Similitud mediante el coseno del ángulo entre los dos vectores en el modelo vectorial

A partir de la *Figura 5*, se comprueba como en el numerador de la fórmula se realiza el producto escalar de los vectores (del documento y la consulta) tal y como se ha visto en el apartado anterior, mientras que en el denominador se realiza el producto de los módulos de ambos vectores. Al contrario del producto escalar, esta medida de similitud oscilará entre 0 y 1 ya que se trata del coseno del ángulo. Manteniendo el mismo ejemplo que en el producto escalar, se comprueba cuáles son los resultados aplicando esta medida de similitud en la *Figura 6*:

Documento 1: "…las películas se ruedan en escenas donde aparecen actores y actrices…"
Documento 2: "…en las películas actúan varios actores y actrices con distintos roles…"

Conjunto de términos K = {películas, escenas, actores, actrices, roles}
D1 = (5,3,2,2,0)
D2 = (3,0,1,1,4)
Q = (0,0,1,1,2)

CosSim(D1, Q) = 4 / √ (25 + 9 + 4 + 4) • (1 + 1 + 4) = 0,25
CosSim(D2, Q) = 10 / √ (9 + 1 + 1 + 16) • (1 + 1 + 4) = 0,78

Figura 6: Ejemplo de similitud mediante el coseno del ángulo entre los dos vectores

Como se puede comprobar en la *Figura 6*, esta medida de similitud arroja los mismos resultados que la anterior del producto escalar. Es decir, según los pesos asignados de forma aleatoria, el documento 2 es mucho más relevante para la consulta que el documento 1 tanto con el producto escalar como con el coseno. En algunos casos, es posible que ambas medidas no coincidan en resultados. Normalmente, la medida más fiable para analizar la similitud es el coseno del ángulo entre vectores.

2.1.2.3 PESO DE LOS TÉRMINOS

En todos los ejemplos que se han visto hasta ahora, los pesos han sido asignados de forma aleatoria para ver el funcionamiento de cada modelo. Pero existen diversas técnicas para calcular los pesos de los términos según el documento en el que se encuentren, para que así, los resultados a las consultas sean realmente relevantes. Una de esas técnicas es la conocida como frecuencias **tf-idf**. A continuación se explican cada una de estas medidas.

La medida **tf (term frequency)** es la frecuencia de un término i en un documento j. **Mide la relevancia** de un documento para un término basada en la representatividad de ese término en el documento. Por lo que, simplemente, tf es el número de veces que aparece un término en un documento.

$$tf_{ij} = freq_{ij}$$

La medida **idf (inverse document frequency)** es la frecuencia de un término i en el resto de la colección de documentos que se tengan. **Mide la eficacia de un término** en búsquedas basadas en su poder de diferenciación con otros documentos del corpus. En el caso de que un término aparezca en muchos documentos del corpus, tendrá un valor discriminativo pequeño. Por lo que, idf es el número de documentos de la colección en los que aparece un término. Se calcula de la siguiente forma:

$$idf_{ij} = \log(N / (n_j + 1))$$

En la fórmula anterior, N representa el número total de documentos del corpus y n_j el número de documentos del corpus en los que aparece el término. Como se puede apreciar, en el denominador se suma 1 para los casos en los que n_j es 0 y así evitar obtener un error en el logaritmo.

Estas dos medidas se suelen combinar para obtener el peso de un término i en un documento j, como se puede observar en la *Figura 7*. Además, es habitual utilizar tf x idf para ver qué documento es más relevante para una consulta dada.

$$\mathbf{W_{i,j} = tf_{i,j} \times Idf_i = freq_{i,j} \times \log (N / (n_j + 1))}$$

Figura 7: Fórmula para calcular el peso de un término en un documento con tf-idf

2.1.2.4 VENTAJAS Y DESVENTAJAS DEL MODELO VECTORIAL

Las principales ventajas del modelo vectorial son las siguientes:

- Tiene en cuenta tf/idf y la longitud del documento.
- Grado de relevancia y matching parcial.
- Mejores resultados en experimentos, sobre todo en grandes colecciones.

Por otro lado, la principal desventaja del modelo vectorial es el tiempo de cálculo, más aún si se calcula la similitud a través del coseno.

2.1.3 MODELO PROBABILÍSTICO

El **modelo probabilístico** calcula la probabilidad de que un documento sea relevante para una consulta, teniendo en cuenta los términos que aparecen en ambos. Es un modelo complejo y costoso en lo que se refiere a computabilidad. Sin embargo, es más robusto que los anteriores ya que es puramente teórico. Además, asume la independencia de términos por lo que la probabilidad de york P(york) es la misma que la probabilidad de york o new P(york | new). Al igual que los anteriores modelos, asigna pesos a los términos, pero en este caso se asignan pesos positivos cuando es probable que el documento sea relevante y pesos negativos cuando es probable que el documento no sea relevante.

La idea de este modelo es la siguiente: dada una consulta, existe un conjunto de documentos que contiene los documentos relevantes y no otros. Si se tuviera una descripción adecuada de este conjunto, no habría problema para encontrar los documentos, pero no se tiene. Por lo que el modelo presupone que existe ese conjunto de documentos relevantes (R). Los documentos que no aparecen en este conjunto se consideran no relevantes (R').

Teniendo en cuenta que la probabilidad de que el documento d sea relevante se representa P(R|d) y que la probabilidad de que el documento d no sea relevante se representa P(R'|d), un documento será relevante si: P(R|d) > P(R'|d). Además, la similitud entre un documento y una consulta es la siguiente:

$$Sim(d, q) = P(R|d) / P(R'|d)$$

Partiendo de que un documento se caracteriza por los términos de los que consta, se puede considerar P(k|R) la probabilidad de que el término k se encuentre en el conjunto de documentos relevantes. Por lo que P(R|d), podría estimarse como la agregación de las probabilidades de los términos que componen el documento d. Sin embargo, no se conoce R, se tiene que averiguar. Una posibilidad es realizar algunas suposiciones previas y después refinar los resultados. Por ejemplo, se puede suponer inicialmente una P(k|R) igual para todos los términos. Los documentos de esta recuperación inicial servirían para aproximar R.

2.1.3.1 VENTAJAS DEL MODELO PROBABILÍSTICO

Las ventajas más destacadas del modelo probabilístico son las que se indican a continuación:

- Ordena los resultados por relevancia.
- Sigue un razonamiento matemático basado en probabilidades, lo que permite que tenga extensiones populares como las redes bayesianas.

2.1.3.2 DESVENTAJAS DEL MODELO PROBABILÍSTICO

En cuanto a las principales desventajas del modelo probabilístico, cabe destacar las siguientes:

- Es poco intuitivo y se obtienen unos resultados muy pobres.
- No es posible conocer al principio el conjunto de documentos relevantes.

2.2 LENGUAJES DE RECUPERACIÓN DE LA INFORMACIÓN

En esta sección se van a repasar los principales lenguajes de consulta existentes para recuperar información de bases de datos relacionales, de la web semántica y de bases de datos orientadas a grafos.

2.2.1 SQL

SQL (Structured Query Language) es el lenguaje estándar ANSI/ISO de definición, manipulación y control de **bases de datos relacionales**. Se trata de un lenguaje declarativo, es decir, sólo hay que indicar qué se desea hacer (mediante sentencias) y no cómo hacerlo. Además, al tratarse de un estándar, este lenguaje es válido (exceptuando algunas diferencias en la sintaxis y los tipos de datos existentes) para numerosos sistemas gestores de bases de datos (MySQL, SQL Server, Oracle, entre otros).

Las sentencias SQL se dividen en dos categorías. Por un lado se encuentra el **lenguaje de definición de datos (Data Definition Language o DDL)**, el cual permite la modificación de la estructura de los objetos que conforman la base de datos, mientras que por el otro está el **lenguaje de manipulación de datos (Data Manipulation language o DML)**, que permite recuperar y trabajar con los datos almacenados. En las próximas secciones, se va a detallar cada una de estas dos categorías tomando como referencia lo que se explica en [2].

2.2.1.1 LENGUAJE DE DEFINICIÓN DE DATOS (DDL)

En este lenguaje se incluyen todas aquellas sentencias con las que se crean, borran y modifican las tablas y demás objetos de la base de datos. Principalmente, existen cuatro sentencias: CREATE, DROP, ALTER y TRUNCATE. Seguidamente se procede a detallar cada una de ellas.

- **CREATE:** esta sentencia se utiliza para la creación de objetos en la base de datos, como por ejemplo, las tablas de una base de datos, las vistas, aserciones, dominios, procedimientos almacenados, disparadores, índices, etc.

- **DROP:** se utiliza para borrar objetos de la base de datos, como por ejemplo, las tablas, vistas, índices, aserciones, dominios, etc. Por ejemplo, para borrar una tabla de la base de datos, simplemente con ejecutar la sentencia **DROP TABLE nombre_tabla;** se borraría la tabla correspondiente. Asimismo, para borrar la base de datos entera, sería **DROP DATABASE nombre_baseDatos;**.

- **ALTER:** con esta sentencia se puede modificar la estructura de una tabla u objeto. Es decir, se pueden añadir o quitar campos a una tabla, se puede cambiar el tipo de un campo, modificar disparadores, etc.

Seguidamente se muestra un ejemplo, en el que se modifica la tabla *ACTOR* creada anteriormente, añadiendo un nuevo campo denominado *NACIONALIDAD*, con su correspondiente tipo de datos.

```
ALTER TABLE "ACTOR" ADD "NACIONALIDAD" VARCHAR2(30 BYTE);
```

Figura 8: Modificando tabla "ACTOR" con SQL en Oracle

- **TRUNCATE:** esta sentencia elimina todos los registros de una tabla. Cabe destacar que no acepta la cláusula WHERE, por lo que será útil sólo cuando se quieran borrar todos los datos de una tabla. Es importante resaltar que con esta sentencia se vacía la tabla que se indique, pero esta sigue existiendo en la base de datos después de ejecutar. A priori, parecería una sentencia perteneciente a DML pero no es así ya que, internamente, la sentencia TRUNCATE borra la tabla y la crea de nuevo, sin ejecutar ninguna transacción. La sentencia completa para vaciar una tabla es **TRUNCATE TABLE nombre_tabla;**.

2.2.1.2 LENGUAJE DE MANIPULACIÓN DE DATOS (DML)

En este lenguaje se incluyen todas aquellas sentencias con las que se manipulan o recuperan los datos. Las principales sentencias son: SELECT, INSERT, UPDATE y DELETE. A continuación se especifica cada una de ellas.

- **SELECT:** mediante esta sentencia se pueden consultar los datos almacenados de una o varias tablas de la base de datos. La forma básica de esta sentencia es la que se muestra en la siguiente figura.

```
SELECT [ALL | DISTINCT] <campo> [{, <campo>}]
FROM <nombreTabla> | <nombreVista> [{,<nombreTabla> | <nombreVista>}]
[WHERE <condición> [{ AND|OR <condición>}]]
[GROUP BY <campo> [{, <campo>}]]
[HAVING <condición> [{ AND|OR <condición>}]]
[ORDER BY <campo> [ASC|DESC] [{, <campo> [ASC|DESC]}]]
[LIMIT <númeroFilasARecuperar>]
```

Figura 9: Forma básica de la sentenia SELECT en SQL

Todo el texto incluido entre corchetes en la *Figura 9* es opcional. A continuación se detallan los principales elementos:

- **SELECT:** es la palabra clave a través de la cual se indica que se quiere consultar a la base de datos.
- **ALL:** indica que se quieren recuperar todos los valores, aunque haya algunos repetidos varias veces. Es el valor por defecto y en muy pocas ocasiones se utiliza.
- **DISTINCT:** indica que sólo se quieren recuperar valores distintos. Es decir, indica que se eliminen aquellos valores que estén repetidos, mostrándolos una única vez.
- **FROM:** indica la tabla o las tablas de donde se quieren recuperar los datos. En el caso de que se incluyan varias tablas, sería una "consulta combinada" o "join" y habría que aplicar una condición de combinación en la cláusula WHERE. Esto se verá algo más adelante.
- **WHERE:** esta cláusula permite incluir una condición que se debe cumplir para que las tuplas sean recuperadas. Si hay filas que no cumplen dicha condición, no serán devueltas. Admite los operadores lógicos AND y OR.
- **GROUP BY:** esta cláusula permite agrupar los resultados obtenidos por uno de los campos. Se usa junto a las funciones de agregación (AVG, COUNT, MAX, MIN, SUM, etc.).

- **HAVING:** esta cláusula siempre se utiliza junto con GROUP BY y permite incluir una condición, aplicada sobre los campos referentes a GROUP BY, que se debe cumplir para que los datos sean devueltos.
- **ORDER BY:** permite ordenar los resultados por un campo. Se puede ordenar de forma ascendente (ASC) o descendente (DESC), aunque la forma predeterminada es la primera.
- **LIMIT:** con esta cláusula se consigue recuperar el número de filas que se indique. Si la consulta recupera 1000 tuplas pero se ha especificado la cláusula LIMIT 500, sólo se mostrarán las 500 primeras filas recuperadas.

- **INSERT:** esta sentencia se utiliza para añadir registros a una única tabla de la base de datos. La forma básica de esta sentencia es la que se muestra en la siguiente figura.

```
INSERT INTO nombreTabla (columna1 [, columna2, …])
VALUES (valor1 [, valor2, …]);
```

Figura 10: Forma básica de la sentencia INSERT en SQL

- **UPDATE:** esta sentencia se utiliza para actualizar los valores que se deseen de un conjunto de registros. La forma básica de esta sentencia es la siguiente.

```
UPDATE nombreTabla
SET columna1 = valor1 [, columna2 = valor2, …]
WHERE columnaX = valorX;
```

Figura 11: Forma básica de la sentencia UPDATE en SQL

- **DELETE:** mediante esta sentencia se elimina uno o varios registros de una tabla. La forma básica de esta sentencia es la que se muestra a continuación.

```
DELETE FROM nombreTabla
WHERE columna1 = valor1;
```

Figura 12: Forma básica de la sentencia DELETE en SQL

En el Anexo B incluido al final de este documento, se añaden algunos ejemplos de uso de las cláusulas citadas en este apartado, además de otras sentencias importantes a la hora de trabajar con SQL.

2.2.2 SPARQL

SPARQL es un lenguaje de consulta sobre grafos RDF en la web semántica [5]. Este lenguaje ha jugado un papel esencial en los avances experimentados por la web semántica hasta el momento. A través de SPARQL, se podrán construir consultas que obtengan resultados de distintas fuentes de datos. Pero antes de profundizar un poco más sobre este lenguaje de recuperación de la información, se va a realizar una pequeña introducción sobre la web semántica, para comprender los conceptos clave que se deben conocer para poder entender el funcionamiento de este lenguaje sin dificultades.

2.2.2.1 LA WEB SEMÁNTICA

La **web semántica** es una **red de datos** que pueden ser procesados directa o indirectamente por máquinas [6]. Es una web extendida que permite trabajar de forma conjunta a seres humanos y máquinas. Los usuarios en Internet, obtendrán respuestas a sus preguntas de manera mucho más rápida y sencilla gracias a la web semántica. Esto es debido a que la **información** se encuentra mucho mejor **definida, estructurada y conectada**. En la web semántica, los datos pasan a ser información dotada de significado.

Debido a la gran cantidad de recursos existentes hoy en día en Internet, se originan ciertos problemas que la web semántica trata de solucionar. Los principales problemas son la sobrecarga de información y el contraste entre las distintas fuentes de información. La web semántica intenta eliminar estos dos problemas autorizando al software a que realice tareas como el procesamiento del contenido web.

Para que la web semántica funcione de forma eficiente es imprescindible que los contenidos de las webs estén correctamente etiquetados y descritos. Para esto, el lenguaje HTML no es suficiente, por lo que se necesitan otros lenguajes semánticos más potentes capaces de representar el conocimiento por medio de metadatos y ontologías.

Cuando surgieron los problemas comentados anteriormente, sólo se utilizaba el lenguaje XML para representar la información. A través de XML existen numerosas estructuras para representar la misma información, lo cual causa bastantes inconvenientes a la hora de recuperarla. Por esto, la **W3C (World Wide Web Consortium)** ha estandarizado el proceso mediante el Marco de Descripción de Recursos, **RDF (Resource Description Framework)**, el cual permite especificar recursos y las relaciones existentes entre ellos siempre con la misma estructura. De esta manera, una información sólo estará representada de una única manera y se facilitará, en gran medida, el proceso de recuperación y acceso a la información. RDF fragmenta el conocimiento en afirmaciones formadas por la estructura *sujeto-atributo-valor* denominadas **tripletas**.

La web semántica permite navegar a través de datos y semánticas, en lugar de hiperenlaces. Pero para ello, se deben cumplir una serie de requisitos que se indican a continuación:

1. **Identidad**: todos los elementos de una tripleta se deben poder identificar por medio de URIs. El valor puede ser una URI o un literal (por ejemplo texto, número o fechas).
2. **Accesibilidad**: las tripletas deben ser accesibles por protocolos, como por ejemplo http. Las URIs también pueden ser expresadas a través de prefijos (PREFIX dc: http://purl.org/dc/elements/1.1/).
3. **Estructura**: se han de emplear lenguajes normalizados y públicos, como RDF y SPARQL.
4. **Navegación**: la utilidad reside en el número de recursos enlazados por RDF. Cuantos más recursos haya enlazados, mayor será el potencial de la web semántica.

Sin embargo, la web semántica tiene aún importantes problemas a solucionar. Entre ellos, se encuentran los siguientes:

- Las técnicas de Procesamiento del Lenguaje Natural no consiguen todavía descifrar la semántica, por lo que de momento, no se puede comprender la información.

- Existe sobrecarga de información.
- Heterogeneidad de fuentes de información.

2.2.2.2 SPARQL ENDPOINTS

Mediante SPARQL se puede acceder a información existente en la web por medio de los conocidos endpoints. Un SPARQL endpoint es el interfaz que proporcionan los datasets de RDF (conjunto de tripletas enlazadas entre sí, las cuales serán consultadas) para ejecutar las consultas SPARQL. Existen endpoints genéricos, a los que hay que indicarles el dataset que se quiere utilizar, y endpoints vinculados directamente a un dataset en concreto. Los resultados arrojados por los endpoints pueden tener diferentes formatos (HTML, JSON, XML, RDF/XML, etc.).

DBpedia es una plataforma que transforma a RDF cada una de las tripletas que se introducen en la Wikipedia. Es decir, DBpedia genera información semántica a partir de la Wikipedia. Concretamente, el proceso encargado de la extracción de esta información, lo hace de 15 de los idiomas de Wikipedia (entre ellos el español). Para cada uno de estos idiomas, el comité de internalización de DBpedia ha asignado un SPARQL Endpoint [7]. En cada uno de ellos, están disponibles todas las tripletas correspondientes según el idioma. El SPARQL endpoint **Virtuoso SPARQL Query Editor**[1] permite ejecutar consultas al grafo de la DBpedia (versión en español).

2.2.2.3 TIPOS DE CONSULTAS CON SPARQL

Las **consultas en SPARQL**, habitualmente están **formadas por** conjuntos de **tripletas**. Estas tripletas son muy parecidas a las de RDF, con la diferencia de que en estas tripletas, uno de los tres elementos (sujeto, atributo o valor) pueden ser una variable. Es decir, que los datos que se estén buscando (variables) pueden ser el sujeto, atributo o valor de cada una de las tripletas que conformen la consulta.

SPARQL ofrece 4 tipos distintos de consultas que son los siguientes:

- **SELECT**: recupera los datos que se estén buscando.
- **CONSTRUCT**: devuelve un grafo RDF. Es decir, devuelve un conjunto de tripletas enlazadas entre sí.
- **ASK**: devuelve un valor booleano (verdadero o falso) indicando si se ha encontrado o no una correspondencia con el patrón incluido en la consulta.
- **DESCRIBE**: recupera un grafo RDF que indica una descripción de los recursos localizados.

2.2.2.4 SINTAXIS CONSULTAS TIPO "SELECT"

De los 4 tipos de consultas que se han especificado en el apartado anterior, la que habitualmente se utiliza es la de tipo SELECT. Por ello, se va a detallar cada una de las cláusulas que puede tener este tipo de consultas, indicando la función que atañe a cada una de ellas. Antes de nada, cabe destacar que este tipo de consultas se utilizan cuando se necesitan obtener datos en concreto. Las cláusulas que pueden formar parte de consultas tipo *SELECT* son:

[1] http://es.dbpedia.org/sparql

- **Prefijos**: son los espacios de nombre (namespaces) y qnames con los que se realizará la consulta.
- **SELECT**: esta cláusula es obligatoria y en ella se indican los campos que se quieren recuperar por medio de variables. Estas variables se identifican por medio de un nombre (es recomendable que tenga semántica con los valores devueltos para ese campo, para que la lectura de los resultados sea más sencilla) y un cierre de interrogación (?) antes del nombre. Por lo que si se quisiese recuperar el nombre de los países de la Unión Europea, una posible variable sería *?nombrePaís*.
- **FROM**: es opcional. Sólo se incluye en el caso de que sea un endpoint genérico, para indicar dónde se encuentra el dataset al que se quiere preguntar.
- **WHERE**: es una cláusula obligatoria. En ella se incluyen las tripletas RDF. Al final de cada una de las tripletas se pone un punto (.). Al menos uno de los elementos de cada tripleta será una variable.
- **FILTER**: es una cláusula opcional que se incluye dentro del WHERE para filtrar los datos y conseguir recuperar aquellos en los que se está realmente interesado.
- **ORDER BY**: es opcional. Permite ordenar los resultados obtenidos por una de las variables. Se puede ordenar tanto de manera ascendente (ASC) como descendente (DESC). Se incluye después de la cláusula WHERE.
- **LIMIT**: este elemento es también opcional y en él se indica el número de resultados que se desean obtener al ejecutar la consulta. Se incluye después de las cláusulas WHERE y ORDER BY (si está presente en la consulta).

Seguidamente, se van a analizar distintos ejemplos de consultas de este tipo para comprobar la funcionalidad de cada una de estas cláusulas. Dichas consultas se deberán ejecutar contra el grafo de la DBpedia, utilizando el SPARQL endpoint indicado anteriormente.

```
PREFIX res: <http://dbpedia.org/resource/>
PREFIX dbpedia2: <http://dbpedia.org/property/>
Select ?pagina ?tipo ?idioma ?propietario ?creador where {
res:Gmail dbpedia2:language ?idioma.
res:Gmail dbpedia2:type ?tipo.
res:Gmail foaf:homepage ?pagina.
res:Gmail dbpedia2:owner ?propietario.
res:Gmail dbpedia2:author ?creador.
}
```

Figura 13: Consulta SPARQL - Información sobre el concepto GMAIL

Como se puede apreciar en la *Figura 13*, en primer lugar se indican los prefijos que se utilizarán en la consulta. Estos prefijos son prefijos de namespaces y su objetivo es facilitar la generación de la consulta, evitando que se tenga que escribir constantemente, en cada una de las tripletas de la consulta, la url del namespace. De esta forma, tenemos los prefijos *res* y *dbpedia2*. A continuación se indica la cláusula *SELECT,* indicando las variables que se quieran devolver. Por último, en la cláusula WHERE, se incluyen las tripletas (en negrita) con las que se selecciona la propiedad que se quiera recuperar y se guarda en la variable correspondiente.

Para entender correctamente la sintaxis de las tripletas, se va a analizar la primera que aparece en la consulta anterior. Como su propio nombra indica, cada tripleta consulta de tres partes. En la primera se indica el recurso del que se va a querer extraer información. En este

caso, se utiliza el prefijo *res* indicando que el recurso será *Gmail*. En la segunda parte, se selecciona la propiedad en la que se esté interesado del recurso seleccionado anteriormente. En el ejemplo, se utiliza el prefijo *dbpedia2* seguido de la propiedad *language*. Por último, en la última parte de la tripleta, se indica la variable en la que se devolverá el valor de esa propiedad del recurso seleccionado. Es conveniente recordar que cada tripleta acaba en punto.

Además, cabe destacar que para realizar estas consultas se debe conocer cómo está estructurada la información de cada recurso para utilizar los prefijos y las propiedades correctamente. De este modo, como se puede observar en el ejemplo anterior, se conocen cada una de las propiedades del recurso al que se consulta. Para finalizar con este ejemplo, cabe resaltar que a la hora de seleccionar una propiedad de un recurso en una tripleta, se puede utilizar un prefijo o directamente incluir la propiedad de la misma manera en la que se encuentra codificada en el recurso. Un ejemplo se encuentra en la tercera tripleta de la consulta anterior, donde se selecciona la propiedad directamente (*foaf:homepage*).

Por lo que la consulta de la *Figura 13* recupera información sobre el concepto **Gmail**, en concreto recupera la homepage, el tipo de recurso asociado, los idiomas en los que se encuentra disponible el recurso, el propietario y el creador del sitio web mail.google.com. El resultado obtenido de la ejecución de la consulta, en el SPARQL endpoint indicado anteriormente, es el siguiente.

pagina	tipo	idioma	propietario	creador
https://www.gmail.com/mail/help/about.html	http://dbpedia.org/resource/Webmail	72	http://dbpedia.org/resource/Google	http://dbpedia.org/resource/Paul_Buchheit
https://mail.google.com/	http://dbpedia.org/resource/Webmail	72	http://dbpedia.org/resource/Google	http://dbpedia.org/resource/Paul_Buchheit

Figura 14: Resultado ejecución consulta Figura 13

En el siguiente ejemplo, no se conoce el recurso sobre el que se quiere consultar, pero a partir de un valor conocido de una propiedad del recurso, se obtiene el mismo en una de las tripletas de la consulta, utilizándose para el resto de ellas. En este caso, se recuperan los mismos datos que en el anterior exceptuando la página principal (ya que es conocida) y en cuyo lugar se devuelve la url del recurso.

```
PREFIX dbpedia2: <http://dbpedia.org/property/>
Select ?x ?tipo ?idioma ?propietario ?creador where {
?x dbpedia2:language ?idioma.
?x dbpedia2:type ?tipo.
?x foaf:homepage <https://mail.google.com/>.
?x dbpedia2:owner ?propietario.
?x dbpedia2:author ?creador.
}
```

Figura 15: Consulta SPARQL - Preguntar a partir de tripleta conocida

En la *Figura 15* se comprueba que no se conoce el recurso sobre el que se quiere preguntar, pero sí se conoce la *homepage*. Por lo que, con la tercera tripleta, se obtiene el recurso en *?x*, el cual se utilizará en el resto de tripletas. Como se quiere devolver el recurso, se debe incluir en la cláusula *SELECT* la variable *?x*. Además, a partir de la *Figura 15*, se comprueba que en una tripleta puede haber dos variables. A continuación se muestra el valor devuelto en *?x*.

x
http://dbpedia.org/resource/Gmail

Figura 16: Recurso devuelto consulta Figura 15

Otra forma de obtener los mismos resultados que en las consultas anteriores, consiste en la utilización de filtros. En la siguiente consulta se va a utilizar el filtro *regex(?x, "gmail", "i"),* con el cual se indica que sólo se quieren los registros en los que la variable ?x contenga el texto "gmail". Con el tercer parámetro ("i"), se indica que no importan las mayúsculas/minúsculas, es decir que no es "case sensitive". Si se quisiera realizar un filtro "case sensitive" sólo habría que incluir los dos primeros parámetros.

```
PREFIX dbpedia2: <http://dbpedia.org/property/>
Select ?x ?pagina ?tipo ?idioma ?propietario ?creador where {
?x dbpedia2:language ?idioma.
?x dbpedia2:type ?tipo.
?x foaf:homepage ?pagina.
?x dbpedia2:owner ?propietario.
?x dbpedia2:author ?creador
FILTER regex(?x, "gmail", "i")
}
```

Figura 17: Consulta SPARQL - Con filtro "regex"

A partir de la *Figura 17*, se comprueba que no se conoce ni el recurso ni ninguna de sus propiedades como ocurría en el ejemplo anterior. Lo que sí se sabe es que se quieren buscar recursos con la palabra clave "gmail". Es por esto, que se utiliza el filtro comentado anteriormente, para obtener sólo los dos resultados que se obtenían en la *Figura 14*. Es importante remarcar que esta consulta tarda más en ejecutar que las anteriores, ya que primero se obtienen todas las tripletas y después se filtran.

En el ejemplo que se muestra a continuación, se pretende recuperar qué recursos ha fundado el creador de Gmail (Paul Buchheit).

```
PREFIX onto: <http://dbpedia.org/ontology/>
Select ?autoriaBuchheit where {
<http://dbpedia.org/resource/Paul_Buchheit> onto:knownFor ?autoriaBuchheit.
}
```

Figura 18: Consulta SPARQL - Creaciones de Paul Buchheit

Como se aprecia en la *Figura 18*, en este caso se incluye un prefijo con el namespace *onto: <http://dbpedia.org/ontology/>* en lugar de *dbpedia2: <http://dbpedia.org/property/>.* Este último es un subconjunto del primero, por lo que habrá elementos que sean idénticos en ambos namespaces y otros que sólo pertenecerán a uno de los dos. Como se exponía anteriormente, en SPARQL es esencial conocer cómo se ha codificado el recurso para saber que prefijos de namespaces utilizar. En este caso, se ha analizado el recurso y se ha determinado que es más relevante el criterio de la ontología *knownFor*. Asimismo, en este ejemplo se ha optado por incluir la URI del recurso en la propia tripleta, en lugar de incluir un prefijo como se hacía en las consultas anteriores. El resultado obtenido de la ejecución de la consulta es el que se muestra a continuación.

autoriaBuchheit
http://dbpedia.org/resource/Google
http://dbpedia.org/resource/Gmail
http://dbpedia.org/resource/FriendFeed

Figura 19: Resultado ejecución consulta Figura 18

Con la siguiente consulta se busca recuperar el resumen de la vida de Paul Buchheit sólo en español (se encuentra codificado en varios idiomas). Además, con ella se intenta mostrar otro tipo de filtro que se puede utilizar con SPARQL. En concreto, se trata del filtro *lang(?abstract) = "es"*. Con él, se consigue recuperar sólo el resumen en español de la vida de Paul Buchheit.

```
PREFIX res: <http://dbpedia.org/resource/>
PREFIX dbpedia2: <http://dbpedia.org/property/>
PREFIX onto: <http://dbpedia.org/ontology/>
Select ?creador ?abstract where {
res:Gmail dbpedia2:author ?creador.
?creador onto:abstract ?abstract
FILTER (lang(?abstract) = "es")
}
```

Figura 20: Consulta SPARQL - Con filtro lang

En la *Figura 20*, se observa el filtro comentado anteriormente, en el que la variable ?abstract se filtra indicando que sólo se desean recuperar los resúmenes en español ("es"). Igualmente, se aprecia cómo se incluyen dos tripletas. La primera de ellas consigue recuperar, en la variable ?creador, el nombre de Paul Buchheit, el cual es utilizado a su vez en la segunda tripleta como recurso, para poder obtener su resumen a través de la ontología *abstract*. Esto se podría reducir a una única tripleta ya que se conoce el recurso Paul Buchheit. Cabe destacar, en el caso de que se quisieran recuperar resúmenes tanto en español como en inglés, simplemente habría que concatenar dos filtros mediante el operador OR:

FILTER ((lang(?abstract) = "es") || (lang(?abstract) = "en"))

Finalmente, se van a detallar dos ejemplos más de consultas con SPARQL algo más complejas que las anteriores. La consulta que se muestra a continuación devuelve 6 creadores de recursos, junto con su fecha y lugar de nacimiento.

```
PREFIX onto: <http://dbpedia.org/ontology/>
Select distinct ?creador ?fecha_nacimiento ?lugarNacimiento where {
?recurso onto:author ?creador .
?creador onto:birthDate ?fecha_nacimiento.
?creador onto:birthPlace ?lugarNacimiento.
}
LIMIT 6
```

Figura 21: Consulta SPARQL - Otros creadores de recursos

A partir de la *Figura 21*, se comprueba cómo se recopilan todos los creadores de recursos con la primera tripleta, mientras que con las dos siguientes se recuperan sus respectivas fechas y lugares de nacimiento. De la misma forma, se evidencia que se puede utilizar la cláusula *distinct* al igual que en SQL, para no devolver un mismo resultado varias veces. Además, para devolver únicamente 6 de todos los creadores de recursos recuperados, se añade la cláusula *LIMIT 6,* depués del cierre de llave correspondiente a la cláusula WHERE.

Con esta última consulta, se pretende mostrar otra manera de identificar en una tripleta un recurso no conocido. Hasta ahora, cuando no se conocía un recurso, se utilizaba una variable en la primera parte de la tripleta. Sin embargo, también se puede utilizar [] cuando no se sabe el recurso en una tripleta. A continuación se incluye un ejemplo.

```
PREFIX dbpedia2: <http://dbpedia.org/property/>
Select distinct ?creador ?hijo where {
[] dbpedia2:author ?creador.
?creador dbpedia2:child ?hijo.
}
```

Figura 22: Consulta SPARQL - Identificando recurso no conocido con []

A partir de la *Figura 22*, se comprueba cómo se utiliza la apertura y cierre de corchetes ([]) cuando no se conoce el recurso en una tripleta. Como en este caso se quieren recuperar todos los creadores, independientemente del recurso que sea, en la primera tripleta se utiliza [], indicando que el recurso debe tener la propiedad *author* para poder recuperar en la variable *?creador* la persona en cuestión. En la segunda tripleta, se utiliza la variable *?creador* como recurso, para obtener los hijos de cada uno de los creadores en la variable *?hijo*. Por lo que, con este último ejemplo de consulta SPARQL, se recuperarían los distintos hijos de los creadores de recursos.

2.2.3 CYPHER

Cypher es el lenguaje de consulta en grafos de Neo4j (Base de Datos Orientada a Grafos que se verá más adelante en este documento). Se trata de un lenguaje de consulta declarativa, inspirado en SQL, con el que se pueden describir patrones en grafos. Además, este lenguaje está diseñado para que su legibilidad y expresividad sean máximos. Permite describir qué se quiere seleccionar, insertar, actualizar o eliminar de una base de datos en grafo sin la necesidad de que se le indique cómo hacerlo exactamente [8].

2.2.3.1 UTILIZANDO CYPHER

Como se verá en los próximos capítulos, un grafo es un conjunto de **nodos** conectados entre sí a través de **relaciones**. Los datos a almacenar serán los nodos en el grafo. Es decir, los nodos serán entidades como por ejemplo actores, películas, libros, etc. Tanto los nodos como las relaciones pueden tener atributos que almacenen información adicional sobre ellos. A estos atributos se les denomina **propiedades**. Además, se pueden agrupar varios nodos según la temática que se desee mediante las **etiquetas**.

Seguidamente, se profundiza sobre estos conceptos utilizando la sintaxis de Cypher. Se comienza con un patrón simple para reflejar cómo representar los nodos y relaciones en este lenguaje de consulta:

```
(a) – [:ACTÚA_EN] –> (b)
```

Figura 23: Ejemplo de patrón con Cypher

Como se observa en la *Figura 23*, para representar nodos en Cypher se indica el nombre que se desee entre paréntesis. Por lo que en el ejemplo, se tendrían los nodos (a) y (b). En el caso de las relaciones, se representarán con la sintaxis – [] –> siempre entre dos nodos. En los corchetes se indicará el nombre de la relación que une a esos nodos precedido de dos puntos (:). En el ejemplo anterior, se indica que se pretenden recuperar los nodos de los cuales salga una relación *:ACTÚA_EN* hacia cualquier otro nodo.

Cuando se quiere hacer referencia a la propiedad de un nodo, se debe seguir el siguiente patrón:

```
(a {nombre: "Santiago Segura"})
```

Figura 24: Ejemplo de propiedad de un nodo (Cypher)

Como se puede observar en la *Figura 24*, dentro de la declaración del nodo (dentro de los paréntesis) se incluye una apertura y cierre de llaves ({}) en las que se escribe el nombre de la propiedad (en este caso sería la propiedad "nombre") acompañado de dos puntos (:) y el valor de la propiedad indicada entre comillas. Esto será muy útil cuando se pretendan realizar consultas para que sean más legibles y no tener que utilizar la cláusula WHERE.

Para señalar el tipo de etiqueta de los nodos, se indica de la siguiente forma:

```
(a: Persona)
```

Figura 25: Ejemplo de etiqueta (Cypher)

En la *Figura 25* se comprueba que para reflejar que se quieren nodos de una determinada etiqueta se escriben dos puntos después del nombre que se le haya asignado al nodo y a continuación, el nombre de la etiqueta. Las etiquetas son muy útiles para acotar en gran medida el número de nodos a los que visitar cuando se ejecuten las consultas.

Por lo que, con lo comentado hasta ahora, se podrían conseguir patrones como el que se muestra a continuación:

```
(a:Persona) - [:ACTÚA_EN] -> (b:Película)
```

Figura 26: Personas que han actuado en alguna película (Cypher)

En la *Figura 26* se puede ver cómo sería el patrón en el caso de que se quisieran seleccionar a todas aquellas personas (etiqueta *Persona*) que hayan actuado (relación *ACTÚA_EN*) en alguna película (etiqueta *Película*). Además, cabe destacar que tanto *a* como *b* son identificadores, es decir, se pueden poner los identificadores que se deseen, no varía en nada la ejecución de la consulta. Por lo que, en vez de a y b, se podrían tener los identificadores ***actor***:*Persona* y ***película***:*Película* para que el patrón sea aún más auto-descriptivo.

Una vez que se han repasado los elementos básicos para formar patrones en Cypher, se va a continuar con las consultas. En Cypher, existen dos cláusulas obligatorias en todas las consultas. Estas dos cláusulas son las siguientes:

1. **MATCH**: todas las consultas empezarán siempre por esta cláusula indicando a continuación el nodo o los nodos que se quieren recuperar mediante los correspondientes patrones.
2. **RETURN**: todas las consultas finalizarán con esta cláusula que servirá para indicar qué datos se quieren recuperar de los nodos o relaciones devueltos.

A continuación, se van a mostrar algunos ejemplos de consultas sencillas:

```
MATCH (n) RETURN n;
```

Figura 27: Consulta que devuelve todos los nodos del grafo (Cypher)

En la *Figura 27* se puede ver la consulta que nos devolvería cualquier nodo que exista en el grafo, por lo que se obtendría el grafo completo. Al devolver *RETURN n* se devuelven los nodos, es decir, también se recuperan todas aquellas propiedades que tenga cada uno de

ellos. Cuando se ejecuta una consulta, Neo4j permite visualizar el resultado tanto en forma de grafo como en forma de filas devueltas. En la siguiente figura, se puede observar el resultado, en forma de grafo, de ejecutar esta consulta sobre un pequeño grafo creado previamente:

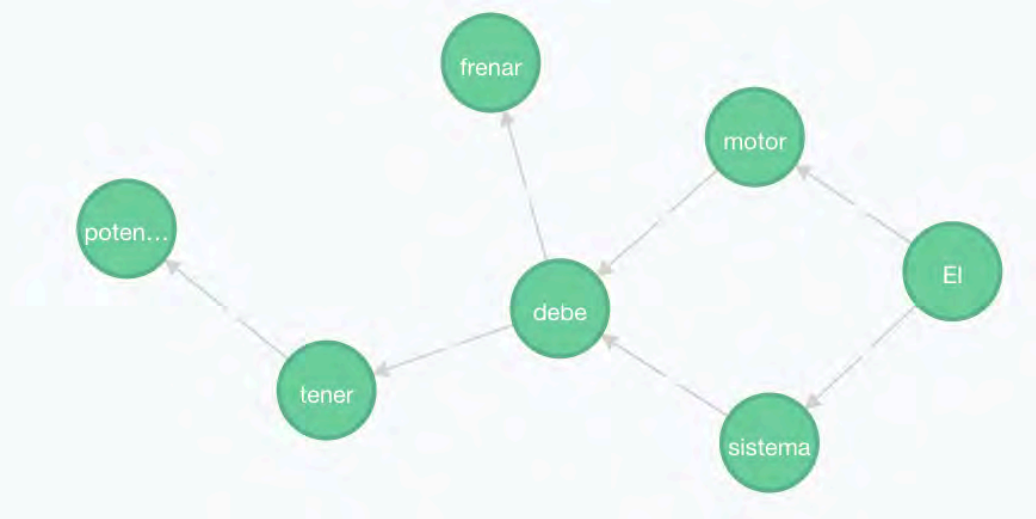


Figura 28: Grafo resultante de la consulta de la Figura 27

Otra sencilla consulta podría ser la siguiente:

```
MATCH (n) --> (m) RETURN n, m;
```

Figura 29: Consulta que devuelve todos los pares de nodos con relación de n a m (Cypher)

Con la consulta de la *Figura 29* se obtendrían todos aquellos pares de nodos que tengan una relación del nodo n al nodo m. En la figura que vemos a continuación, se muestra el resultado, en filas devueltas[2], de ejecutar la consulta de la *Figura 29* sobre el mismo grafo que la consulta de la *Figura 27*:

[2] No se muestra el resultado completo, sólo se incluyen las 4 primeras filas devueltas.

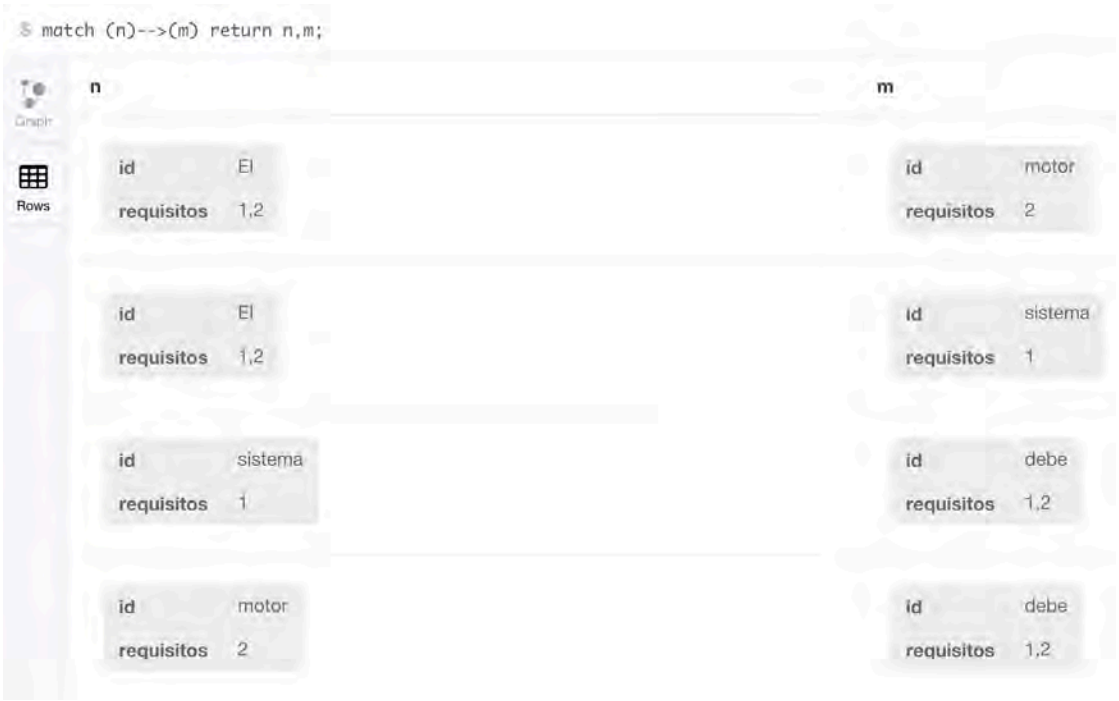


Figura 30: Resultado de la ejecución de la consulta de la Figura 29

Volviendo al ejemplo anterior de actores y películas, si se incluye el patrón, que se veía en la *Figura 26,* en una consulta, quedaría de la siguiente forma:

```
MATCH (a:Persona) - [:ACTÚA_EN] -> (b:Película)
RETURN a,b;
```

Figura 31: Consulta sobre "Grafo de Películas" (Cypher)

Como se puede comprobar en la *Figura 31*, se recuperan tanto los actores como las películas en las que han participado. El resultado en forma de filas devueltas sería el siguiente[3]:

Figura 32: Resultado ejecución consulta Figura 31

Asimismo, también se pueden devolver las propiedades que se quiera de cada uno de los nodos en lugar de los nodos completos como se ha hecho hasta ahora. En el siguiente

[3] Sólo se muestran las dos primeras filas devueltas.

ejemplo de la *Figura 33,* se devuelven las propiedades *nombre* y *título* de los nodos recuperados:

```
MATCH (a:Persona) - [:ACTÚA_EN] -> (b:Película)
RETURN a.nombre,b.título;
```

Figura 33: Devolviendo propiedades con Cypher

Como se comprueba en la *Figura 33,* simplemente lo único que se ha cambiado es la cláusula *RETURN* en la que se indica que devuelva sólo el nombre de los actores y el título de las respectivas películas. Por lo que se recuperaría una lista con las propiedades que se hayan incluido en la consulta como se observa en la *Figura 34*:

Figura 34: Resultado ejecución consulta Figura 33

Además, al igual que en SQL, se puede incluir la cláusula WHERE en las consultas de Cypher para filtrar por los datos (nodos) que se deseen. Un sencillo ejemplo de consulta incluyendo ésta cláusula podría ser el siguiente:

```
MATCH (a:Persona) - [:ACTÚA_EN] -> (b:Película)
WHERE a.nombre=”Keanu Reeves”
RETURN b.título;
```

Figura 35: Consulta con cláusula WHERE (Cypher)

En la *Figura 35* se puede observar como se mantiene la cláusula *MATCH*, seleccionando todos aquellos nodos que se encuentren unidos por una relación *:ACTÚA_EN.* Pero en este caso, se incluye la cláusula *WHERE* para indicar que sólo se pretenden recuperar las películas en las que ha actuado *Keanu Reeves*. Por lo que, como sólo se devuelven las películas, la lista que se obtiene sería la siguiente:

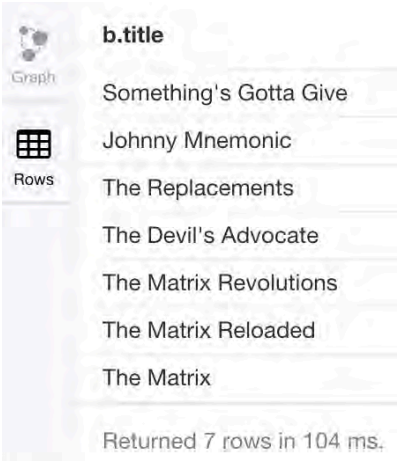


Figura 36: Resultado ejecución consulta Figura 35

2.2.3.2 CAMINOS EN CYPHER

Un camino (path) consiste en una serie de nodos conectados entre sí mediante relaciones. Si se utilizan consultas basadas en caminos es una forma muy sencilla de conseguir datos mucho más concretos que si se ejecuta alguna de las que se han visto hasta el momento.

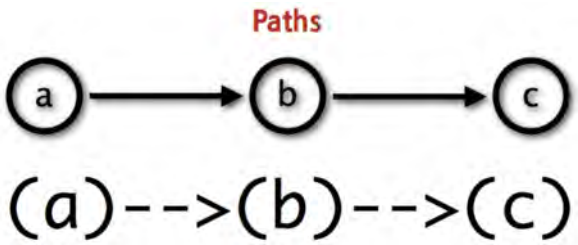


Figura 37: Camino en Cypher[4]

Seguidamente, se muestran algunas consultas en las que se utilizan caminos.

```
MATCH (actor) - [:ACTÚA_EN] -> (película) <- [:DIRIGE] - (director)
RETURN actor.nombre, película.título, director.nombre;
```

Figura 38: Consulta basada en caminos (Cypher)

Como se puede observar en la *Figura 38*, para incluir un camino en la cláusula *MATCH,* lo que se hace es enlazar dos patrones en uno sólo. A continuación, se analiza por partes:

- **Primera parte ((actor) - [:ACTÚA_EN] -> (película)):** con esta primera parte del patrón se obtienen los actores y las películas en las que han participado. Esto sería como los patrones que ya se han visto. Para formar el camino, se incluye la segunda parte que se detalla a continuación.
- **Segunda parte ((película) <- [:DIRIGE] - (director)):** con esta segunda parte se indica que también se pretenden conocer los directores de las películas seleccionadas en la primera parte del patrón.

[4] http://neo4j.com/graphacademy/online-course/

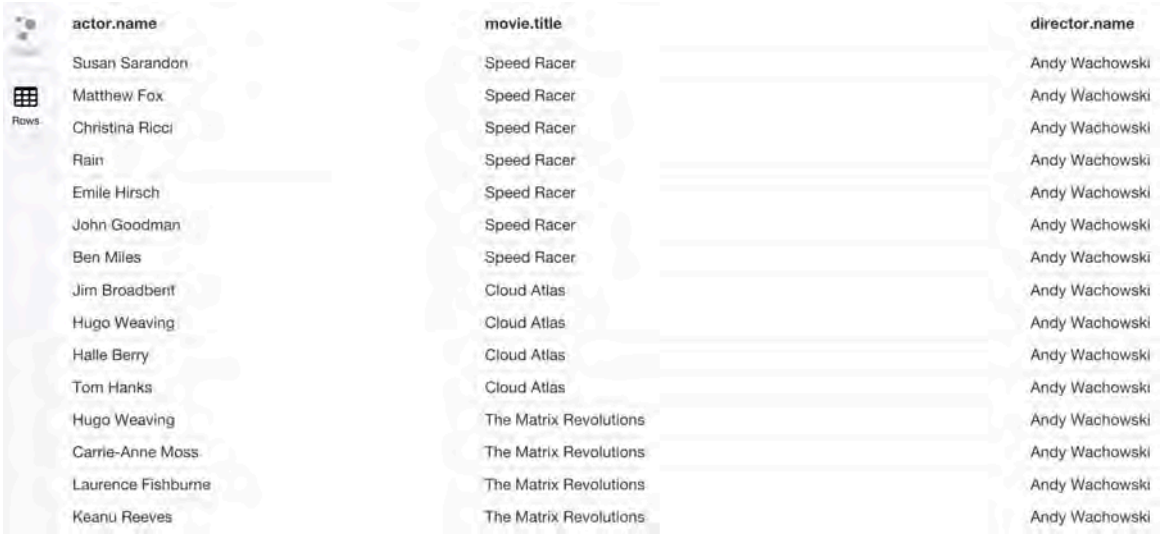
Por último, en la cláusula *RETURN* se recupera el nombre de los actores, directores y los títulos de las correspondientes películas. Por lo que, un posible enunciado para esta consulta podría ser el siguiente: "*Recuperar todos los directores con los que ha trabajado cada actor, mostrando también el título de las correspondientes películas"* o este otro *"Recuperar todos los actores con los que ha trabajado cada director, mostrando también el título de las correspondientes películas".*

Cabe destacar, que este tipo de consultas admiten diferentes notaciones. Por lo que, la consulta de la *Figura 38* podría escribirse también de la siguiente forma obteniendo el mismo resultado:

```
MATCH (actor) - [:ACTÚA_EN] -> (película),
(director) - [:DIRIGE] -> (película)
RETURN actor.nombre, película.título, director.nombre;
```

Figura 39: Distinta notación consulta Figura 38

El resultado de ejecutar cualquiera de las dos anteriores consultas serían tres columnas, la primera de ellas con los nombres de los actores, la segunda con los títulos de las películas y la última con los nombres de los directores como se puede ver en la *Figura 40*[5]:



actor.name	movie.title	director.name
Susan Sarandon	Speed Racer	Andy Wachowski
Matthew Fox	Speed Racer	Andy Wachowski
Christina Ricci	Speed Racer	Andy Wachowski
Rain	Speed Racer	Andy Wachowski
Emile Hirsch	Speed Racer	Andy Wachowski
John Goodman	Speed Racer	Andy Wachowski
Ben Miles	Speed Racer	Andy Wachowski
Jim Broadbent	Cloud Atlas	Andy Wachowski
Hugo Weaving	Cloud Atlas	Andy Wachowski
Halle Berry	Cloud Atlas	Andy Wachowski
Tom Hanks	Cloud Atlas	Andy Wachowski
Hugo Weaving	The Matrix Revolutions	Andy Wachowski
Carrie-Anne Moss	The Matrix Revolutions	Andy Wachowski
Laurence Fishburne	The Matrix Revolutions	Andy Wachowski
Keanu Reeves	The Matrix Revolutions	Andy Wachowski

Figura 40: Resultado ejecución consultas Figuras 38 y 39

Adicionalmente, también se puede nombrar un camino cuando sea incluido en una consulta. Simplemente se tiene que igualar el camino a una variable y devolverla, mediante la cláusula RETURN. A continuación se incluye un ejemplo:

```
MATCH p = (actor) - [:ACTÚA_EN] -> (película) <- [:DIRIGE] - (director)
RETURN p;
```

Figura 41: Nombrando caminos con Cypher

Como se puede observar en la *Figura 41,* en la cláusula *MATCH* de la consulta, se iguala el patrón, que en este caso representa un camino, a la variable p, devolviendo esta última en la cláusula *RETURN*. El resultado de ejecutar esta consulta es el que aparece a continuación:

[5] Sólo se muestra una pequeña parte del total de filas recuperadas.

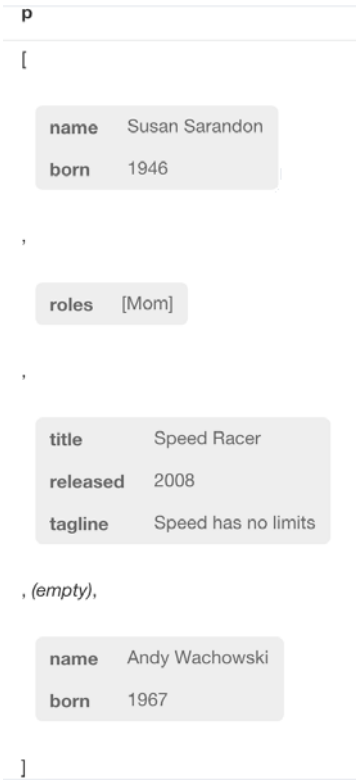


Figura 42: Resultado ejecución consulta Figura 41

Como se puede comprobar en la *Figura 42*, cuando se iguala un camino a una variable (p en este caso) y se devuelve esta, Neo4j recupera por cada una de las filas que se veían en la *Figura 40,* los siguientes valores:

- El nodo correspondiente al actor/actriz.
- Las propiedades que tenga la relación *ACTÚA_EN,* que conecta el nodo del actor/actriz con el de la película (en este caso, se recupera el rol con el que participa la actriz en la película).
- El nodo correspondiente a la película.
- Las propiedades que posea la relación *DIRIGE,* que conecta al nodo de la película con el del director (en este caso, esa relación no incluye ninguna propiedad).
- Por último, el nodo que refleja las propiedades del director.

El resultado que se muestra en esta última figura, concierne únicamente a la primera fila devuelta. Por cada fila devuelta, Neo4j devolvería los datos que se han identificado anteriormente (sólo se ha incluido la primera fila a modo de ejemplo). Por lo que, si se va a ejecutar una consulta basada en un camino y se quieren recuperar todos y cada uno de los datos que existan tanto en los nodos afectados como en las relaciones, una forma sencilla de conseguirlo es nombrando el camino y devolviendo esa variable.

Además, en el caso de que sólo se quisiesen recuperar los nodos del camino, en vez de devolver la variable p, se tendría que devolver **nodes(p)**. La consulta sería la misma que la mostrada en la *Figura 41* excepto la cláusula *RETURN,* que en vez de devolver p, devolvería nodes(p). Si por el contrario, lo que se quieren recuperar son las relaciones, con sus respectivas propiedades, que existan a lo largo del camino, se incluirá en la cláusula *RETURN* **rels(p)**.

Como aspecto más avanzado, Cypher permite describir caminos de longitud variable mediante una estrella (*) indicada en la relación. A continuación se detallan las principales nomenclaturas utilizadas para este tipo de caminos:

- **(a) - [*1..4] -> (b)**: este patrón indica que se busquen nodos desde *a* atravesando relaciones hasta una profundidad máxima de 4 relaciones.
- **(a) - [*] -> (b)**: con este patrón se buscarían nodos desde *a* hasta cualquier profundidad. Es decir, su buscarían nodos por todo el grafo.
- **(a) - [*3] -> (b)**: si se quieren buscar nodos a profundidad 3, se debería emplear este patrón.
- **(a) - [:AMIGOS*2] -> (b)**: con este patrón se buscarían nodos a profundidad 2 y al ser la relación *AMIGOS*, se recuperarían amigos de amigos del nodo *a*.

Por lo que, en resumen, los caminos son muy útiles para conseguir información precisa y, con las funcionalidades de manipulación de caminos que nos ofrece Cypher, podremos obtener realmente la información que deseemos en cada momento, recuperando nodos, relaciones o ambos, de la forma que se ha visto a lo largo de esta sección.

2.2.3.2.1 Función SHORTESTPATH()

Cabe destacar que Neo4j dispone de la función ***shortestPath()***. Esta función permite encontrar el camino más corto entre dos nodos. A continuación se muestra una consulta a modo de esquema para la utilización de esta función.

```
MATCH p = shortestPath( (nodo1) - [*] - (nodo2) )
RETURN length(p), nodes(p);
```

Figura 43: Función shortestPath()

En la *Figura 43*, se puede comprobar que la función *shortestPath()* se indica en la cláusula *MATCH* igualándola a una variable (en este caso *p*) e incluyendo entre los paréntesis de la función el patrón que se desee. En este caso, se devuelve la longitud del camino mínimo encontrado y los nodos integrantes de dicho camino. En el supuesto de que sólo se quisiese mostrar el nombre de los nodos que conformen el camino, en la cláusula RETURN se tendría que utilizar la función ***extract()*** de la siguiente forma.

```
RETURN extract( n in nodes(p) [1..-1] | n.nombre );
```

Figura 44: Función extract()

Como se puede observar en la *Figura 44*, en los paréntesis de la función *extract()* se indica el identificador n para cada uno de los nodos del camino, entre corchetes se indica que el primer y último nodos del camino no se tengan en cuenta y finalmente se indica la propiedad que se quiera mostrar por cada nodo recuperado.

Para finalizar esta sección, se incluye una consulta utilizando estas dos funciones para comprobar cómo se aplican a un ejemplo concreto.

```
MATCH (keanu:Persona {nombre:"Keanu Reeves"}),
(kevin:Persona {nombre:"Kevin Bacon"})
MATCH p = shortestPath( (keanu) - [:CONOCE*] -> (kevin) )
RETURN extract( n in nodes(p) [1..-1] | n.nombre );
```

Figura 45: Utilizando las funciones shortestPath() y extract()

A la hora de implementar la consulta existen varias alternativas. Una de ellas es como se observa en la *Figura 45*, donde primero se seleccionan los dos nodos involucrados, que serán los extremos del camino mínimo que se pretende encontrar, después con una nueva cláusula *MATCH*, se incluye la función *shortestPath()* con el patrón correspondiente (en este caso se indica la relación *CONOCE*, lo que quiere decir que se pretende encontrar el camino mínimo entre los dos nodos atravesando sólo relaciones de ese tipo) y finalmente, se devuelve la propiedad *nombre* de los nodos involucrados en el camino (exceptuando los extremos). Básicamente, el enunciado podría ser *"Recuperar el camino mínimo entre Keanu Reeves y Kevin Bacon atravesando únicamente relaciones del tipo CONOCE"*.

2.2.3.3 OTRAS CLÁUSULAS Y CONSULTAS AVANZADAS EN CYPHER

Al igual que en SQL existen numerosas cláusulas que se pueden aplicar en las consultas para filtrar u ordenar los resultados, en Cypher también se encuentran disponibles muchas de esas cláusulas. A continuación se incluyen las más importantes con algunos ejemplos.

1. **Cláusula WHERE**: como se ha visto en algún ejemplo anterior, con esta cláusula se puede filtrar por la propiedad de un nodo o relación. En Cypher, existen dos posibles nomenclaturas que se van a analizar con un par de ejemplos.

```
MATCH (n:Persona)
WHERE n.nombre = "Tom Hanks"
RETURN n;
```

Figura 46: Ejemplo cláusula WHERE (nomenclatura 1) con Cypher

En este primer ejemplo de la *Figura 46*, se indica explícitamente en la consulta la cláusula *WHERE* seleccionando en este caso, que se quiere el nodo que tenga de nombre *Tom Hanks*. La segunda manera de incluir esta cláusula en un consulta de Cypher es como se muestra en la siguiente figura:

```
MATCH (n:Persona {nombre:"Tom Hanks"})
RETURN n;
```

Figura 47: Ejemplo cláusula WHERE (nomenclatura 2) con Cypher

Como se puede comprobar en la consulta que aparece en la *Figura 47,* no se indica explícitamente la cláusula, simplemente en la declaración del nodo, después de la etiqueta *Persona* se incluyen dos llaves en las que se escribe la propiedad por la que se desea filtrar, seguida de dos puntos (:) y el valor que se quiera entre comillas. De esta forma se filtran los resultados de igual forma que en la consulta de la *Figura 46*.

2. **Cláusula ORDER BY**: permite ordenar los resultados obtenidos en base a un campo numérico, tanto de forma ascendente (asc) como descendente (desc). La opción predeterminada para esta cláusula es la ordenación ascendente.

```
MATCH (n:Persona) - [:ACTÚA_EN] -> ()
RETURN n.nombre, n.nacimiento
ORDER BY n.nacimiento;
```

Figura 48: Ejemplo cláusula ORDER BY (Cypher)

En la *Figura 48* se observa un ejemplo de uso de ORDER BY. Esta cláusula se incluye después del RETURN de la consulta. Es muy interesante ya que se pueden obtener distintos resultados en base a fechas de nacimiento, en número de películas en las que ha participado cada actor, etc. En este caso, se ordena de forma ascendente por la fecha de nacimiento, por lo que se recuperan actores que hayan actuado en algo, obteniendo antes a los de mayor edad (no tiene porqué ser una película ya que el nodo al que apunta la relación está vacío. Esto quiere decir que puede ser cualquier nodo siempre que sea apuntado por la relación que se indica).

3. **Cláusula LIMIT**: permite limitar los resultados de una consulta. Al igual que en SQL, cuando no interesa recuperar excesivas filas, se puede acotar ese número con esta cláusula. Seguidamente, se incluye un ejemplo para ilustrar su uso:

```
MATCH (n:Persona) - [:ACTÚA_EN] -> ()
RETURN n.nombre, n.nacimiento
ORDER BY n.nacimiento desc
LIMIT 10;
```

Figura 49: Ejemplo de uso de la cláusula LIMIT (Cypher)

La consulta que se observa en la *Figura 49* permite saber dónde se ubica la cláusula *LIMIT* en una consulta Cypher. Como se puede comprobar, se incluye después del RETURN y también después del ORDER BY (en el caso de que la consulta lo tenga). Con esta consulta se recuperarían los 10 actores más jóvenes que han actuado en algo.

4. **Cláusula SKIP**: permite saltar el número de filas que se quieran del total de filas recuperadas. Cuando se esté analizando los resultados de una consulta, si los primeros resultados devueltos no se quieren tener en cuenta, esta cláusula es ideal. A continuación se incluye un ejemplo para completar la explicación:

```
MATCH (n:Persona) - [:ACTÚA_EN] -> ()
RETURN n.nombre, n.nacimiento
SKIP 10
LIMIT 10;
```

Figura 50: Ejemplo de uso de la cláusula SKIP (Cypher)

Como se observa en la *Figura 50*, la cláusula *SKIP* se incluye también después del *RETURN* de la consulta, pero antes del *LIMIT*. Esta consulta devolvería los actores/actrices que hayan actuado en algo. Como se ha incluido la cláusula *SKIP,* las 10 primeras filas devueltas se obvian y se recuperan las 10 siguientes. Es decir, si la consulta devolviese 50 resultados, los 10 primeros se obviarían (por el *SKIP 10*) y se recuperarían desde la fila 11 hasta la 20 (por el *LIMIT 10*).

5. **Cláusula DISTINCT**: en las consultas que se han visto hasta el momento, se podrían obtener tuplas repetidas en el caso de que un actor haya actuado en más de una película. Para evitar obtener resultados repetidos, se incluye la cláusula DISTINCT dentro del RETURN de la consulta de la siguiente manera:

```
MATCH (n:Persona) - [:ACTÚA_EN] -> ()
RETURN DISTINCT n
ORDER BY n.nacimiento
LIMIT 5;
```

Figura 51: Ejemplo de uso de la cláusula DISTINCT (Cypher)

En la *Figura 51*, se comprueba que la cláusula *DISTINCT* se indica dentro de la cláusula *RETURN.* De esta forma se elimina cualquier posibilidad de obtener tuplas que sean el mismo valor. Con esta consulta se recuperarían los 5 actores de mayor edad que hayan actuado en algo. Con las consultas anteriores a la de la *Figura 51* se podrían estar recuperando, de las 10 filas recuperadas, el mismo actor varias veces, por lo que no se estarían devolviendo realmente los 10 valores distintos esperados.

Seguidamente, se presentan las principales funciones de agregado que se utilizan en este lenguaje de consulta. Estas funciones de agregado facilitan la manipulación de los resultados para después ordenarlos y conseguir los resultados esperados. Es decir, permiten contar las filas que cumplen con el patrón para un determinado valor, devolver la media aritmética del conjunto de valores correspondientes a un valor en concreto, devolver datos sumados, el máximo y el mínimo de un conjunto de valores de un determinado valor, además de otras funcionalidades. A continuación se detallan las principales funciones:

- **Count(x)**: permite contar el número de filas que se recuperan para un determinado valor. En los paréntesis de la función, en lugar de la "x" que existe ahora, se debería indicar la propiedad de la que se quieren contar las filas recuperadas. Por ejemplo, si Keanu Reeves ha actuado en 15 películas y se busca en cuáles han sido, se obtendrían 15 filas. Sin embargo, en el caso de que sólo se quiera saber el número de películas en las que ha participado Keanu, se debería utilizar esta función, ya que permite contar el número de filas seleccionadas por el patrón (el número de películas) y devolver ese valor junto con el nombre de keanu Reeves en una única fila. Más abajo se podrá comprobar su uso y los resultados obtenidos mediante un sencillo ejemplo.
- **Min(x)**: se utiliza para propiedades que sean de tipo numérico (la propiedad correspondiente se indica entre los paréntesis de la función). Permite devolver el valor mínimo para un determinado valor de un nodo recuperado. Por ejemplo, si se recuperan las notas obtenidas por el alumno "y" (nodo y) en la asignatura "z" (nodo z), si se utiliza esta función sólo se devolvería la nota más baja obtenida por ese alumno en dicha asignatura.
- **Max(x)**: al igual que la función anterior, sólo se utiliza para propiedades de tipo numérico pero en este caso devuelve el valor máximo para un determinado valor de un nodo recuperado. La propiedad de la que se quiera extraer el máximo valor se indica entre los paréntesis de la función.
- **Avg(x)**: esta es otra función para propiedades de tipo numérico y devuelve la media aritmética de un conjunto de valores correspondientes a un nodo en concreto.
- **Sum(x)**: esta función es sólo aplicable a propiedades de tipo numérico y recupera la suma de un conjunto de valores que se refieren al nodo que se haya seleccionado. Al igual que las anteriores funciones, la propiedad que corresponda con los valores que se quieren sumar, se indicará en los paréntesis de la función.
- **Collect(x)**: esta función es muy útil para ahorrar en cuanto a número de filas devueltas se refiere. Esta función agrupa en una única fila todos los valores recuperados para un nodo en concreto. Por ejemplo, en vez de devolver 10 filas para indicar las 10 películas

en las que ha actuado Susan Sarandon, se devuelve una sola fila en la que se incluye una colección con todas esas películas (como un array de valores).

A continuación, se van a detallar un par de ejemplos para comprobar cómo se utilizan estas funciones y cómo son los resultados obtenidos.

```
MATCH (actor:Persona) - [:ACTÚA_EN] -> (película)
RETURN actor.nombre, collect(película.título);
```

Figura 52: Ejemplo de uso función de agregación "collect(x)"

A partir de la *Figura 52,* se comprueba que las funciones de agregación se indican directamente en la cláusula *RETURN* como si se devolviese cualquier otra propiedad. Para evidenciar cuál es el formato de los resultados obtenidos utilizando la función *collect(x),* se incluye la siguiente figura:

Figura 53: Resultado obtenido de ejecutar la consulta de la Figura 52

Como se aprecia en la *Figura 53,* cuando se recupera una propiedad mediante la función *collect(x),* se obtiene una colección con todos los valores correspondientes a otro valor seleccionado. En este caso, para cada nombre de actor/actriz, se obtienen, en una única fila, todos los títulos de las películas en las que ha participado, en vez de devolver cada título en una fila distinta (si no se aplicase la función collect(x)).

Seguidamente, se va a mostrar otro ejemplo para comprobar cómo funcionan las demás funciones de agregación indicadas anteriormente.

```
MATCH (actor:Persona) - [:ACTÚA_EN] -> (película)
RETURN actor.nombre, count(película)
ORDER BY count(película) desc
LIMIT 6;
```

Figura 54: Ejemplo de uso función de agregación "count(x)"

En la *Figura 54* se observa un ejemplo de utilización de la función de agregación *count(x)*. Se indica como todas las funciones en la cláusula *RETURN*. Además, como se comentaba anteriormente, cuando se utilizan este tipo de funciones, es realmente interesante realizar una ordenación de los resultados devueltos por la misma función de agregación sobre la misma propiedad que la indicada en el *RETURN*, como se puede comprobar en la figura. De esta forma, se podrá realizar un análisis más inmediato de los resultados obtenidos. A continuación, se muestran los resultados que devolvería la consulta de la *Figura 54*.

actor.name	count(película)
Tom Hanks	12
Keanu Reeves	7
Hugo Weaving	5
Jack Nicholson	5
Meg Ryan	5
Cuba Gooding Jr.	4

Figura 55: Resultado ejecución consulta Figura 54

Para finalizar con esta sección, se van a analizar algunas consultas más complejas, con las que se recuperan datos mucho más precisos que con las consultas vistas hasta el momento.

```
MATCH (keanu:Persona) - [r:ACTÚA_EN] -> (película)
WHERE keanu.nombre="Keanu Reeves"
AND "Neo" IN r.roles
RETURN película.título;
```

Figura 56: Filtrando por atributo de una relación (Cypher)

A partir de la *Figura 56*, se comprueba que también se pueden **filtrar los resultados de la consulta por una propiedad de la relación** (en este caso, la propiedad *roles*). Como se puede observar, al igual que en los nodos se indican los identificadores (*keanu* y *película*), en la relación también se puede incluir un identificador (en este caso *r*). Después, en la cláusula *WHERE* se filtra por la propiedad *roles* de la relación *ACTÚA_EN*, utilizando el identificador de la relación, para que se recuperen sólo los títulos de las películas en los que Keanu Reeves haya actuado con el rol de "Neo".

```
MATCH (tom:Persona {nombre:"Tom Hanks"}) - [:ACTÚA_EN] -> (película),
(a:Persona) - [:ACTÚA_EN] -> (película)
WHERE a.nacimiento < tom.nacimiento
RETURN DISTINCT a.nombre, (tom.nacimiento – a.nacimiento) AS diferenciaEdad;
```

Figura 57: Retornando cálculos con Cypher

Como se puede ver en la *Figura 57*, **Cypher permite incluir en la cláusula *RETURN* cálculos numéricos** que son recuperados por medio del ***alias*** que se indique (en este caso *diferenciaEdad*). En concreto, la consulta selecciona a Tom Hanks y a todos sus compañeros de reparto de todas sus películas que sean mayores que él. Lo que se recupera realmente es el nombre de esos compañeros junto con la diferencia de edad correspondiente en cada caso.

```
MATCH (tom:Persona {nombre:"Tom Hanks"}) - [:ACTÚA_EN] -> (película),
(otro) - [:ACTÚA_EN] -> (película)
WHERE (otro) - [:DIRIGE] -> ()
RETURN DISTINCT otro;
```

Figura 58: Filtrando mediante patrones con Cypher

A partir de la *Figura 58,* se comprueba que **Cypher** también **permite filtrar los resultados incluyendo patrones en la cláusula *WHERE***. Los patrones que se indiquen en esta cláusula tienen la misma nomenclatura que los incluidos en la cláusula *MATCH*. Como se puede observar, en este caso la consulta selecciona a Tom Hanks y a todos sus compañeros de reparto. Sin embargo, en la cláusula *WHERE* se filtran todos esos compañeros y sólo se

recuperan los que además, hayan dirigido algo. Esta cualidad aumenta la potencia que tiene Cypher a la hora de filtrar resultados.

Por último, se incluye un ejemplo de consulta del tipo ***friends-of-a-friend query,*** muy utilizada en el ámbito de las redes sociales. Son consultas que se realizan en numerosas redes sociales para recomendar personas a los usuarios, basándose en los amigos que tienen sus amigos. En este tipo de consultas es dónde Neo4j funciona de forma excepcional.

```
MATCH (tom:Persona {nombre:"Tom Hanks"}) - [:ACTÚA_EN] -> () <- [:ACTÚA_EN] - (amigo),
(amigo) - [:ACTÚA_EN] -> () <- [:ACTÚA_EN] - (amigoDamigo)
WHERE amigoDamigo <> tom AND NOT ((tom) - [:ACTÚA_EN] -> () <- [:ACTÚA_EN] - (amigoDamigo))
RETURN amigoDamigo.nombre, count(amigoDamigo)
ORDER BY count(amigoDamigo) desc
LIMIT 3;
```

Figura 59: Friends-of-a-friend query (Cypher)

En la *Figura 59,* se puede observar la consulta que devolvería los 3 amigos de los amigos de Tom Hanks, que más han coincidido con los amigos de Tom actuando en algo y que además, no hayan coincidido nunca con Tom Hanks en ningún rodaje. Con el fin de analizar la consulta con mayor nivel de detalle, se divide en tres partes que van a ser explicadas detalladamente a continuación:

- **Cláusula MATCH**: en esta parte de la consulta se selecciona a Tom Hanks, a los compañeros que han actuado con él en todas las películas en las que ha participado y a las personas que han sido compañeras de reparto de sus compañeros. Se seleccionan asignando un identificador para cada uno (*tom*, *amigo* y *amigoDamigo*), de manera que en las siguientes cláusulas se puedan utilizar para filtrar resultado y devolver las propiedades que se precisen.
- **Cláusula WHERE**: en esta cláusula lo primero que se comprueba es que los nodos seleccionados en *amigoDamigo* no sea el propio Tom Hanks. Además, se incluye un patrón con el que se selecciona a los compañeros de los compañeros de Tom con los que Tom ha coincidido actuando en algún rodaje, para eliminarlos de los resultados devueltos (ya que se incluye *NOT* antes del patrón).
- **RETURN, ORDER BY y LIMIT**: por último, se devuelve en el *RETURN* el nombre de los amigos de los amigos de Tom, además del número de ocasiones en las que han coincidido con algún amigo de Tom. Adicionalmente, se ordenan los resultados de manera descendente por las coincidencias calculadas en el *RETURN* y se recuperan tan solo las 3 primeras filas.

2.2.3.4 CREANDO Y EDITANDO UN GRAFO EN CYPHER

En esta sección se va a explicar cómo crear un grafo con Cypher y cómo editarlo a la hora de añadir propiedades, nuevos nodos, eliminar nodos, editar relaciones, etc. En primer lugar, se va a mostrar cómo crear un nodo en Neo4j con Cypher.

```
CREATE (santi:Persona {nombre:"Santiago Segura"}) RETURN santi;
```

Figura 60: Creando un nodo con Cypher

Como se puede comprobar en la *Figura 60*, para crear un nodo con Cypher, se utiliza la cláusula ***CREATE***, incluyendo entre paréntesis un identificador cualquiera, además de la etiqueta a la que pertenecerá el nodo cuando sea creado (es opcional) y entre llaves las propiedades que se quieran añadir al nodo (*nombre* en este caso) junto con los

correspondientes valores entre comillas. La cláusula *RETURN* no es obligatoria, ya que sin ella el nodo se crea correctamente, pero es recomendable para recuperar el nodo que se acaba de crear y comprobar que todo ha sido creado de forma correcta. Entre todos los nodos que se registren en un grafo, cada uno de ellos puede tener diferentes propiedades, incluso los nodos que pertenezcan a la misma etiqueta.

En el supuesto de que se quisiese añadir una nueva propiedad al nodo creado con la figura anterior, la forma que Cypher nos permite realizarlo es la que se observa en la siguiente figura:

```
MATCH (p:Persona)
WHERE p.nombre = ”Santiago Segura”
SET p.nacimiento = 1965
RETURN p;
```

Figura 61: Editando un nodo con Cypher

A partir de la *Figura 61*, se observa que la nomenclatura para editar un nodo en Cypher es muy similar a la de las consultas que se veían en la sección anterior. Simplemente habría que seleccionar el nodo con la etiqueta correspondiente, filtrar por la propiedad incluida al crearlo (*nombre* en este caso) y mediante ***SET*** añadir la propiedad que se desee junto con el valor correspondiente. Finalmente, se devuelve el nodo editado mediante la cláusula *RETURN*. En el caso de que se quisiese modificar el valor de una propiedad de un nodo, la consulta sería exactamente la misma que la de la *Figura 61*, indicando la propiedad afectada junto al nuevo valor que se quiera.

Una vez que se tienen al menos dos nodos creados, ya se puede crear una relación entre ellos. Para crear una relación entre dos nodos, simplemente hay que ejecutar la siguiente sentencia.

```
MATCH (santiago:Persona {nombre:”Santiago Segura”}),
(película:Película {título:”Las brujas de Zugarramurdi”})
CREATE (santiago) - [:ACTÚA_EN {papel:”Actor secundario”, rol:”Miren”}] -> (película);
```

Figura 62: Creando una relación con Cypher

Cuando se quiere crear una relación entre dos nodos, lo primero que hay que hacer es seleccionarlos mediante *MATCH* como se puede ver en la *Figura 62*. Seguidamente, mediante *CREATE*, se crea un patrón simple formado por los dos nodos y la relación que los une entre corchetes. Entre estos corchetes, se incluye el tipo de relación (en este caso *ACTÚA_EN*) y entre paréntesis las propiedades que se quieran incluir en la relación (*papel* y *rol* en este caso). Cada relación existente en el grafo puede tener las propiedades que se deseen, sin obligación de que coincidan de unas relaciones a otras.

Al ejecutar la query anterior varias veces, se crearían tantas relaciones como veces la hayamos ejecutado. En cambio, si se utiliza la cláusula ***MERGE*** en lugar de *CREATE,* en el caso de que exista la relación, esta será devuelta, mientras que si no existe, se creará. A continuación se añade un ejemplo de utilización de *MERGE* para evidenciar la nomenclatura correcta:

```
MATCH (santiago:Persona {nombre:"Santiago Segura"}),
(película:Película {título:"Las brujas de Zugarramurdi"})
MERGE (santiago) - [:ACTÚA_EN] -> (película)
RETURN santiago, película;
```

Figura 63: Utilización de MERGE con Cypher

Asimismo, también se puede añadir una relación a varios pares de nodos a la vez. A continuación se va a mostrar una sentencia con la que se creará la relación *CONOCE* entre todos aquellos actores/actrices y directores/as que hayan coincidido en algún rodaje y aún no tengan una relación *CONOCE* entre ellos.

```
MATCH (a) - [:ACTÚA_EN | :DIRIGE] -> () <- [:ACTÚA_EN | :DIRIGE] - (b)
WHERE NOT (a) - [:CONOCE] - (b)
MERGE (a) - [:CONOCE] -> (b);
```

Figura 64: Creando una relación entre varios pares de nodos con Cypher

Como se puede apreciar en la *Figura 64*, en la relación del patrón incluido en la cláusula WHERE no existe dirección. Con esto se indica que la dirección puede ir en ambos sentidos, es decir, si ya existe una relación *CONOCE* de *a* a *b* o de *b* a *a*, no se creará la relación ya que existe en una dirección u otra.

Cuando se quiera modificar una relación entre dos nodos, la manera de conseguirlo es prácticamente la misma que cuando se modifica un nodo. Primeramente, se debe seleccionar la relación que se desee mediante el correspondiente patrón, filtrando en la cláusula *WHERE* por los nodos de los extremos de dicha relación e indicando mediante ***SET*** las propiedades que se vayan a actualizar o añadir. En la próxima figura se muestra un ejemplo de actualización de una propiedad de una relación.

```
MATCH (kevin) - [r:ACTÚA_EN] -> (mystic)
WHERE kevin.nombre = "Kevin Bacon"
AND mystic.título = "Mystic River"
SET r.rol = "Sean"
RETURN r;
```

Figura 65: Editando una relación con Cypher

Para finalizar con esta sección, se va a detallar cómo eliminar nodos con Cypher. Es importante resaltar que para eliminar un nodo, primero hay que eliminar todas sus relaciones. No obstante, en la misma sentencia se pueden eliminar tanto el nodo como sus relaciones. Por lo que, primero se ha de seleccionar el nodo y relaciones mediante el correspondiente patrón, como siempre y después, mediante la cláusula ***DELETE***, se eliminan por medio de los identificadores que se les haya asignado en la cláusula *MATCH*. En la siguiente figura se puede ver un sencillo ejemplo.

```
MATCH (kevin:Persona {nombre: "Kevin Bacon"}) - [r] - ()
DELETE kevin, r;
```

Figura 66: Eliminando un nodo con Cypher

2.2.4 DISCUSIÓN FINAL DE LOS LENGUAJES EXPUESTOS

En el capítulo de *Lenguajes de recuperación de la Información* se ha incluido el lenguaje SQL como principal lenguaje de consulta sobre bases de datos relacionales, el lenguaje SPARQL idóneo para la Web Semántica y el lenguaje de consulta para grafos de Neo4j, Cypher.

Cabe destacar, que además del lenguaje Cypher, existen otros lenguajes para consultar bases de datos basadas en grafos, como por ejemplo el lenguaje *Bounds Language*[6] que ofrece GraphBase o el lenguaje *Predicate Query Language (PQL)*[7] de Infinite Graph.

2.3 SISTEMAS DE ALMACENAMIENTO BASADOS EN GRAFOS

En esta sección se realiza un pequeño repaso sobre la historia de la teoría de grafos, se señalan las principales ventajas de las BDOG y se incluyen los principales sistemas basados en grafos indicando las principales características de cada uno de ellos.

2.3.1 HISTORIA DE LA TEORÍA DE GRAFOS

Cuando se habla de la teoría de grafos, se está hablando de representación de la información, basada en teoría matemática. En este trabajo, que trata sobre sistemas de almacenamiento basados en grafos, era ineludible mencionar a **Leonhard Euler (1707-1783).** Fue un matemático suizo que inventó la mayor parte de la terminología y notación matemática moderna. Además fue pionero en la teoría de grafos. En 1736, creó y resolvió el problema conocido como **"Los 7 puentes de königsberg"**. Este problema fue el primer estudio de la teoría de grafos [9].

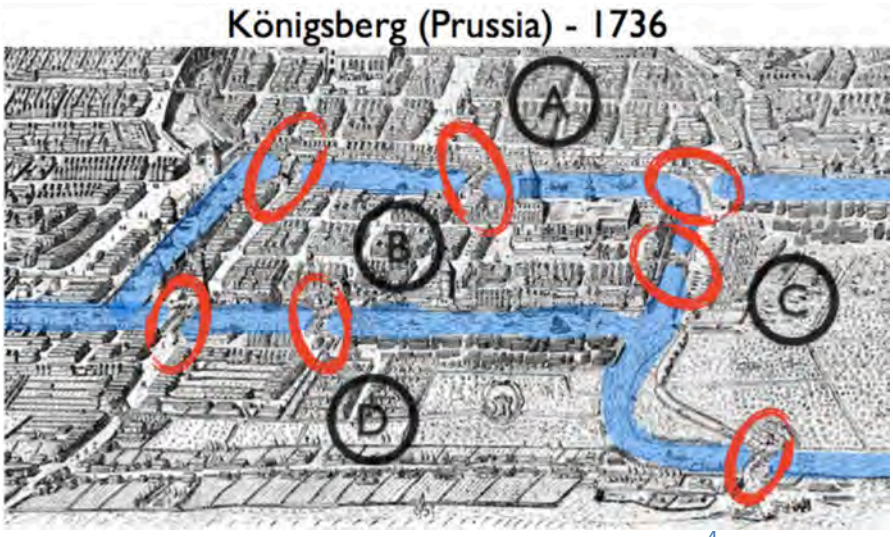


Figura 67: Problema de Los 7 puentes de Königsberg[4]

En la *Figura 67* se puede observar un dibujo representando la ciudad de Königsberg perteneciente a Prussia en 1736. En dicha figura, aparece la zona concreta de la ciudad en la que Euler se basó para formular el problema de "Los 7 puentes de Königsberg". Esta ciudad estaba dividida en 4 zonas separadas cada una de ellas por un río. El problema que planteó Euler es si se podía encontrar un camino partiendo desde un área, pasando por todos los puentes, sin repetir ninguno, para volver a llegar al mismo área de inicio.

[6] http://graphbase.net/JavaAPIHelp.html#BoundsLanguage
[7] http://wiki.infinitegraph.com/3.4/w/index.php?title=PQL/Getting_Started_with_PQL

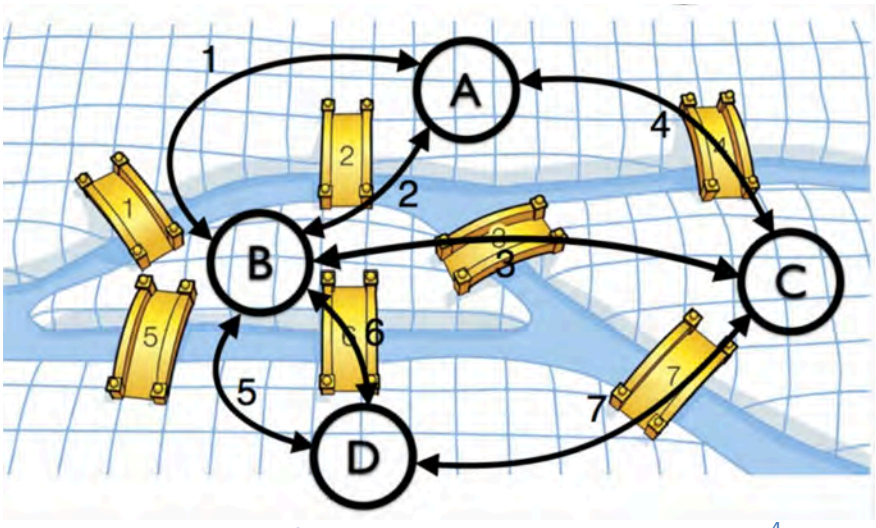


Figura 68: Abstracción del problema de Los 7 puentes de Königsberg[4]

Primeramente, como se puede visualizar en la *Figura* 68, Euler realizó una abstracción del problema identificando cada una de las 4 áreas como nodos y a cada uno de los puentes que unen las áreas como aristas que unen dichos nodos. De esta forma, obtuvo el grafo resultante que observamos más nítidamente en la *Figura 69*.

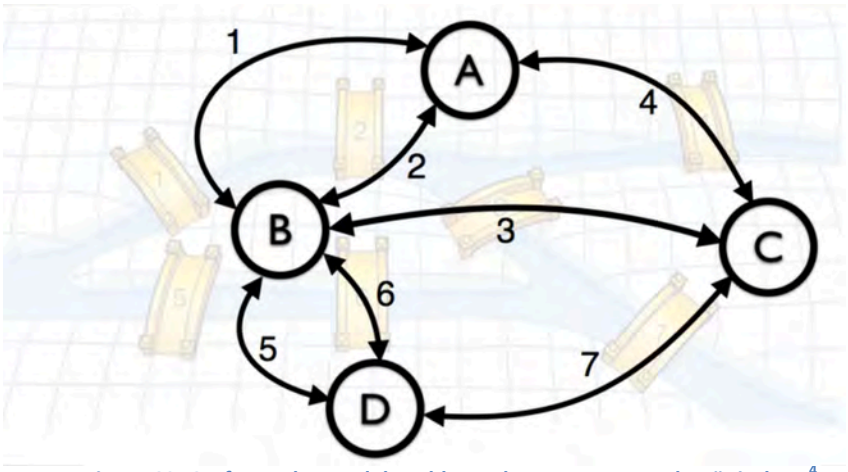


Figura 69: Grafo resultante del problema de Los 7 puentes de Königsberg[4]

Euler indicó que para representar un camino en el que se crucen 7 puentes distintos se necesitan 8 letras. Además, Euler argumentó que al haber 5 puentes que conducen a la isla B, la letra B debía aparecer 3 veces en el supuesto camino para resolver el problema. Asimismo, la letra D tenía que aparecer 2 veces, ya que existen 3 puentes que conducen a D. De la misma forma, A y C debían aparecer 2 veces cada una. Esto quiere decir, que el supuesto camino que se tenía que intentar construir para pasar sólo una vez por todos los puentes y volver al punto de origen, debía constar de 3 veces la letra B y las letras A, C y D cada una 2 veces. Si se suman, se tendría el camino que resolvería el problema formado por 9 letras. Por esto mismo, Euler concluyó que era completamente imposible resolver el problema con una serie de 8 letras y por tanto, no se podían recorrer los 7 puentes de Königsberg de la forma requerida. Sin embargo, Euler no se conformó con encontrar una respuesta al problema sino que formuló una teoría matemática que supuso el comienzo de la teoría de grafos.

Euler analizó de forma exhaustiva si, para cualquier otra configuración en lo que se refiere a número de puentes (aristas) y zonas de tierra (nodos), existía la posibilidad de encontrar el camino inicialmente requerido. Finalmente, concluyó que sí se podía encontrar el camino que buscaba desde el principio, pero sólo si existen como mucho dos nodos en el grafo con valencia impar (siendo la valencia de un nodo el número de aristas que confluyen en ese nodo. En el caso del problema de Königsberg, el número de puentes que confluyen en una porción de tierra). Más concretamente, como se indica en [10], Euler indicó sobre qué grafos se pueden dibujar caminos que sólo pasen una única vez por cada arista:

1. En el caso de que el grafo no tenga nodos de valencia impar, no sólo se podrá dibujar, sino que se podrá empezar el camino en el nodo que se quiera y se finalizará en el nodo que se empezó. Estos caminos cerrados se denominan ciclos y son conocidos como **ciclos o circuitos eulerianos**.

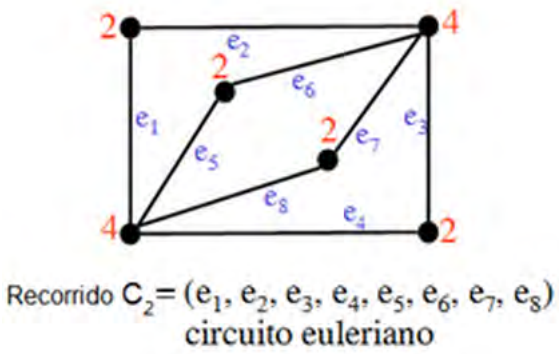


Figura 70: Ejemplo de ciclo euleriano[8]

En la *Figura 70* se observa un ejemplo de ciclo euleriano. El grafo está constituido por 6 nodos, todos ellos con valencia par (cada nodo tiene indicada su valencia). Por lo que, como se ha comentado, se puede realizar un ciclo euleriano. En el *Recorrido C_2* se empezaría en el nodo de abajo a la izquierda, recorriendo las aristas que se indican en C_2 hasta llegar al nodo de origen. Se podría empezar en el nodo que se quisiese y siempre se encontraría un ciclo euleriano.

2. En el caso de que el grafo sólo tenga dos nodos de valencia impar, se podrá dibujar, pero sólo si se empieza en uno de esos dos nodos y se termina en el otro. Esto se conoce como **camino euleriano**.

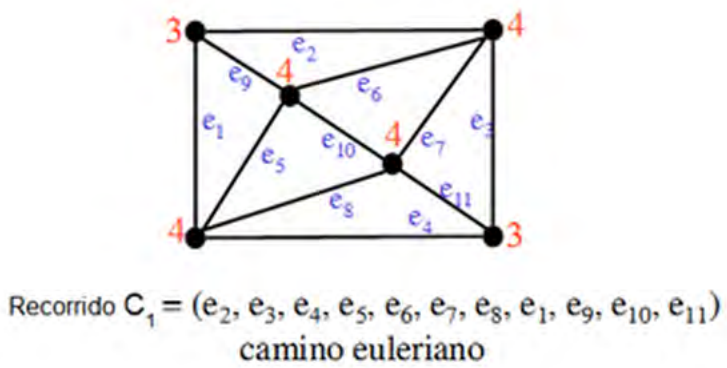


Figura 71: Ejemplo de camino euleriano[8]

[8] https://ensmatejonathan.wordpress.com/2012/06/24/prof-jonathan-brenes/

En la *Figura 71* se observa un ejemplo de camino euleriano. Al igual que la *Figura 70*, el grafo lo componen 6 nodos y cada uno de ellos tiene indicada su valencia. Todos tienen valencia par excepto dos de ellos cuya valencia es 3. Por lo que un camino euleriano sería el que aparece en el *Recorrido* C_1, en el que se comienza en el nodo de arriba a la izquierda (valencia impar) recorriendo las aristas indicadas en dicho recorrido y llegando al otro nodo de valencia impar (abajo a la derecha). Por lo que, tanto el grafo de la *Figura 70* como el de la *figura 71* son **grafos eulerianos**, ya que un grafo se considera euleriano si contiene un ciclo o camino euleriano.

3. En el caso de que el grafo contenga más de dos nodos con valencia impar, no se podrá dibujar ningún camino que satisfaga las condiciones requeridas.

2.3.2 ¿POR QUÉ UTILIZAR BDOG?

A medida que pasan los años, aumenta la demanda de tener almacenada la información y por tanto, la capacidad que se necesita en las aplicaciones que consultan esos datos cada vez es mayor. Normalmente, se requiere analizar e interpretar los datos almacenados, para así, poder generar información útil que sea de provecho para aumentar el valor de una organización.

Con esta situación, como alternativa a las bases de datos tradicionales (modelo relacional), surge la necesidad de encontrar nuevas técnicas que ayuden a realizar esas tareas computacionales de una forma eficiente. Los grafos representan de forma fiel cómo se estructura la información en distintas situaciones de la vida diaria (por ejemplo, en qué medio de transporte ha llegado una persona al trabajo, en qué restaurante ha comido, los amigos que se ha encontrado a lo largo del día, etc.) mediante conceptos y relaciones. Es aquí donde aparecen las BDOG, uno de los tipos de base de datos NoSQL. Mediante los grafos se pueden representar los datos y las relaciones entre los diferentes objetos como un sólo conjunto de datos, de forma que este pueda ser descrito a través de patrones de conocimiento.

En concreto, un grafo es un conjunto de nodos o vértices los cuales están relacionados entre sí mediante relaciones o aristas. Por lo que, un grafo se define de la forma G = (V, E), donde V es el conjunto de vértices del grafo y E el conjunto de aristas. De esta forma, en las BDOG se establecen relaciones binarias entre distintos pares de nodos, donde los nodos se corresponderán con los datos o entidades que se estén manipulando y las aristas indicarán la relación existente entre ambos.

Cabe destacar que una base de datos de este tipo debe estar totalmente normalizada de modo que, pasándolo al modelo relacional se tuviese para cada tabla una sola columna y para cada relación tan sólo dos. De esta forma, se consigue que cualquier alteración en la estructura de la información sólo tenga un impacto local. Además, en las BDOG cada nodo posee un puntero directo a sus nodos adyacentes, lo que significa que no es necesario realizar consultas por índices. Por esto, se dice que una base de datos basada en grafos es cualquier sistema de almacenamiento que permite la adyacencia libre de índice.

Asimismo, conviene mencionar que este tipo de bases de datos no significan que, globalmente, sean mejores que las bases de datos tradicionales. Simplemente, las BDOG funcionan mejor para algunas aplicaciones mientras que para otras su rendimiento es peor. No se trata de elegir un tipo de base de datos para siempre, sino seleccionar la que mejor encaje en función de las características que se necesiten.

Cuando lo que se necesite sea un alto rendimiento permitiendo que sea fácilmente escalable, las BDOG serán una muy buena opción.

Como se refleja en [11], las principales ventajas de las BDOG son:

1. **Rendimiento**: en las bases de datos relaciones, cuando se ejecutan consultas intensivas y con un procesamiento de datos muy alto (por ejemplo, consultas con numerosos joins), el rendimiento baja considerablemente. Sin embargo, en las BDOG tanto el rendimiento como el incremento de los datos, tienden a permanecer constantes. Esto es debido a lo que se conoce como **procesamiento en grafo de forma nativa** (Native Graph Processing). Al tener cada nodo una referencia directa a sus nodos adyacentes, el tiempo que tardará una consulta será sólo proporcional al tamaño de la parte gráfica que tenga que recorrer para satisfacer esa consulta, no de la totalidad del grafo.

2. **Flexibilidad**: en las BDOG no es necesario declarar el tipo de datos para los nodos, ni para las relaciones, ni tampoco para sus respectivas propiedades. Esto permite que se puedan añadir nuevos tipos de relaciones, incluso nuevos nodos o sub-grafos a una estructura ya existente sin modificar las consultas ni la funcionalidad de la aplicación. Esto es a lo que se conoce con el nombre de **grafo aditivo**. Además, tampoco es necesario reservar espacio para las propiedades, como sí hay que hacerlo para los campos de las tablas en las bases de datos relacionales. Esto aumenta aun más la flexibilidad de las BDOG y evita que se desperdicie espacio en los registros almacenados.

3. **Agilidad**: según crece una aplicación, el modelo de datos de la misma deberá evolucionar para adaptarse a las nuevas necesidades que surjan. Las BDOG consiguen esto ya que **se alinean al desarrollo ágil del software**, lo que permite respaldar la evolución de la aplicación para que se acople al ritmo cambiante del entorno de negocio.

2.3.3 MOTORES DE BDOG

En la actualidad, existen numerosos motores de BDOG. A continuación, se indican los más relevantes:

- Neo4j
- KnowledgeMANAGER
- AllegroGraph
- Sparksee
- GraphBase
- Graph Engine
- Infinite Graph
- HyperGraphDB
- OrientDB

Seguidamente, se profundiza sobre los detalles que caracterizan a cada uno de estos motores de BDOG.

2.3.3.1 NEO4J[9]

Neo4j es un software libre de base de datos orientada a grafos escrito en java, que permite almacenar datos de forma estructurada [12]. Se integra perfectamente con otros lenguajes como PHP, Ruby, .Net, Python, Node y Scala. La base de datos se encuentra embebida en un servidor Jetty (servidor http 100% basado en java). Neo4j es compatible con los sistemas operativos Linux, Windows y Mac OS X. Como se veía en el capítulo 2.2.3, el lenguaje de consulta de Neo4j es **Cypher**. Redes sociales y sistemas de recomendación son los campos idóneos para trabajar con Neo4j.

Existen dos versiones de Neo4j. La versión *Community Edition* (open source) y la *Enterprise Edition*. Para realizar pruebas de concepto basta con la versión gratuita pero si se pretende sacar el máximo provecho se debe utilizar la versión EE, que permite realizar monitorización, backups en caliente y sistema de caché de alto rendimiento, entre otras ventajas.

Neo4j no tiene esquema y sus **transacciones** son **ACID** (**A**tomicity, **C**onsistency, **I**solation and **D**urability) [13]. En español serían las características de Atomicidad, Consistencia, Aislamiento y Durabilidad. A continuación se indica lo que significa cada una de ellas:

- **Atomicidad**: propiedad que asegura que la operación se realiza o no se realiza, por lo que, ante un fallo del sistema, no puede quedar a medias. Una operación se considera atómica si no se puede descomponer en pasos intermedios. Por lo que en este caso no habría que controlar nada, excepto comprobar si se realiza o no. Si se tuviera una operación con varios pasos intermedios, esta propiedad dice que se tiene que comprobar que se realizan todos los pasos o ninguno.
- **Consistencia**: propiedad que asegura que todo lo que se empieza se puede acabar correctamente. Es decir, se ejecutarán aquellas operaciones que no rompan las reglas de integridad de la base de datos.
- **Aislamiento**: propiedad que asegura que una operación no afecte a otras. Es decir, asegura que dos transacciones realizadas sobre la misma información sean independientes y sin generar errores.

[9] http://neo4j.com/

- **Durabilidad**: propiedad que asegura que una operación, después de ser realizada, persistirá y no se podrá deshacer aunque falle el sistema. De esta forma se consigue salvaguardar los datos por encima de todo.

2.3.3.1.1 Modelo de datos

El modelo de datos que utiliza Neo4j es un **Grafo de Propiedad (Property Graph).** Un grafo de este tipo está formado por nodos etiquetados y relaciones direccionadas, todos ellos con sus respectivas propiedades. A continuación, se puede ver un ejemplo de un Grafo de Propiedad:

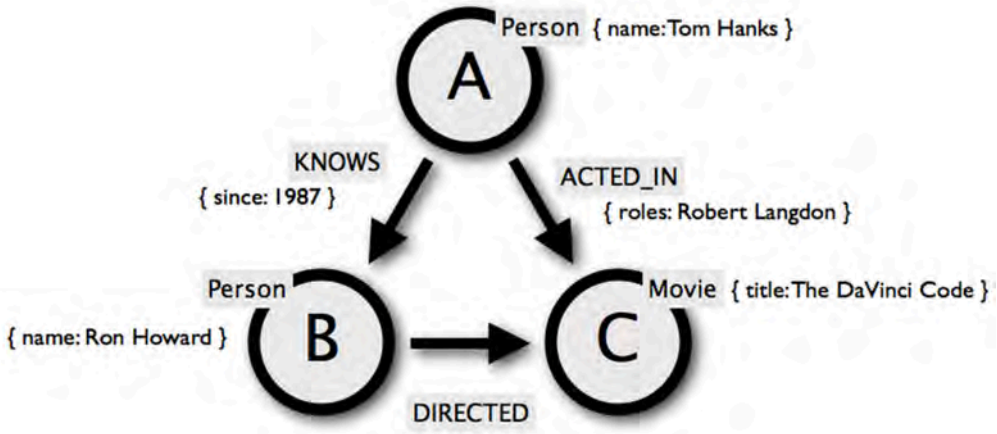


Figura 72: Ejemplo de Grafo de Propiedad[4]

En la *Figura 72* se puede apreciar un grafo compuesto por 3 nodos los cuales se encuentran marcados mediante las etiquetas "Person" y "Movie". Asimismo, se puede observar las tres relaciones existentes entre ellos ("KNOWS", "DIRECTED", "ACTED_IN"), todas ellas con una dirección determinada. Además, tanto los nodos como las relaciones poseen sus respectivas propiedades. Es, por tanto, un sencillo ejemplo de grafo de propiedad.

2.3.3.1.2 Ventajas

1. Neo4j ofrece la capacidad de gestionar grandes cantidades de datos de forma eficiente. Más concretamente, su capacidad se acerca a los 34.000 millones de nodos, 34.000 millones de relaciones, 68.000 millones de propiedades y 32.000 tipos de relaciones.
2. Igualmente, una de sus principales ventajas es la alta velocidad en la ejecución de consultas. Permite saltar millones de veces entre los nodos en tan sólo unos segundos. Esto es posible gracias a lo que se comentaba en el capítulo 2.3.2, cuando se hablaba del **procesamiento en grafo de forma nativa**. Por lo que el almacenamiento está optimizado para consultas sobre datos próximos partiendo de uno o varios nodos, más que para consultas globales a la base de datos.
3. Como BDOG que es, otra ventaja de Neo4j es su modelo de datos flexible, que permite crear nodos y relaciones sin la necesidad de declarar el tipo de datos de ellos.
4. Además, Neo4j es open source o software libre, lo que significa que es muy accesible para todos.
5. Se pueden ejecutar las consultas directamente a través de un API Rest, lo cual facilita su integración con las aplicaciones web.

6. Otra ventaja es cómo almacena los nodos, relaciones y propiedades Neo4j. Realiza un **almacenamiento en grafo de forma nativa** (Native Graph Storage). Esto quiere decir que Neo4j guarda los nodos en un fichero, las relaciones en otro y las propiedades de ambos en otro. Por lo que, al tener las propiedades separadas en otro fichero, el almacenamiento de nodos y relaciones se preocupa exclusivamente de la estructura del grafo. De esta forma, se pueden obtener rápidamente nodos en memoria en base a su id, ya que se conoce en qué posición se encuentran estos. Esta forma de almacenamiento, junto con el procesamiento en grafo de forma nativa hacen que Neo4j proporcione un alto rendimiento cuando se ejecuten consultas sobre grandes volúmenes de datos.

2.3.3.1.3 Desventajas

1. Problemas de compatibilidad: las bases de datos NoSQL, tienen pocas normas en común.
2. Neo4j no dispone de gestión de usuarios. Es decir, no posee sistemas de autenticación. Aunque sí existe una extensión para añadir esta funcionalidad.
3. Falta de experiencia: para algunas empresas, no se encuentra aún lo suficientemente maduro.
4. Neo4j Community no es escalable para conjuntos de datos muy grandes. Hay que cambiar la configuración de la memoria en Neo4j según las necesidades del grafo.
5. Neo4j prefiere tener todo el grafo en memoria, si se puede, pero siempre existen algunas partes del grafo que se mantienen invariables y que es innecesario e ineficiente tenerlas cacheadas. Lo más recomendable es guardar en memoria sólo el conjunto de datos que vaya a ser consultado.
6. Lo que más ocupa en memoria es el almacenamiento de los nodos, por lo que si se guarda demasiada información en cada uno de ellos y no se modela bien el problema, las consultas bajarán considerablemente su rendimiento, volviéndose ineficientes.

2.3.3.1.4 Escenarios de utilización

Existen dos formas de utilizar Neo4j en las aplicaciones. Una de ellas es la posibilidad de que Neo4j esté embebida en dicha aplicación, mientras que la otra sería que Neo4j funcionase como un servidor. A continuación se detallan cada una de ellas:

- **Neo4j embebida en una aplicación:** se pueden incluir las librerías de Neo4j en una aplicación para aprovechar sus ventajas sin tener que depender de un servidor externo. Esta opción equivale a trabajar con la base de datos en memoria. Embeber la base de datos en una aplicación también se puede efectuar de dos formas distintas. La decisión de utilizar una u otra dependerá de las necesidades de rendimiento y de los recursos que disponga el usuario. A continuación, se describen cada una de ellas:

    1. **Utilizar una instancia simple de Neo4j:** si se está en un entorno con pocos recursos (memoria, CPU, etc.) o las necesidades de la aplicación no son demasiado exigentes, esta opción es la adecuada. A esta instancia simple se accede mediante llamadas al **API GraphDatabaseService**[10].

[10] http://neo4j.com/api_docs/2.0.3/org/neo4j/graphdb/GraphDatabaseService.html

2. **Utilizar múltiples instancias:** cuando se necesite una alta disponibilidad de los datos y se disponga de numerosos recursos se podrá utilizar el **API HighlyAvailableGraphDatabase**[11], el cual facilitará la creación de múltiples instancias para cumplir los objetivos.

- **Neo4j funcionando como servidor:** es la forma más habitual de acceder a la base de datos. Funciona como un servidor típico de bases de datos, corriendo en una o varias máquinas (según la necesidades de rendimiento que tenga el usuario), que atiende las peticiones de los clientes. En este caso se trabaja con **peticiones REST** al servidor, para que resuelva las consultas. En el manual que ofrece Neo4j, se incluye un capítulo en el que se explica el **API REST**[12] que utiliza.

2.3.3.1.5 Casos de Uso

En este apartado se repasan los principales casos de uso en los que Neo4j se adapta perfectamente [14]:

1. **Redes y Centro de Datos:** las capas física, virtual y de aplicación de una red, están perfectamente modeladas en un grafo Neo4j integral.

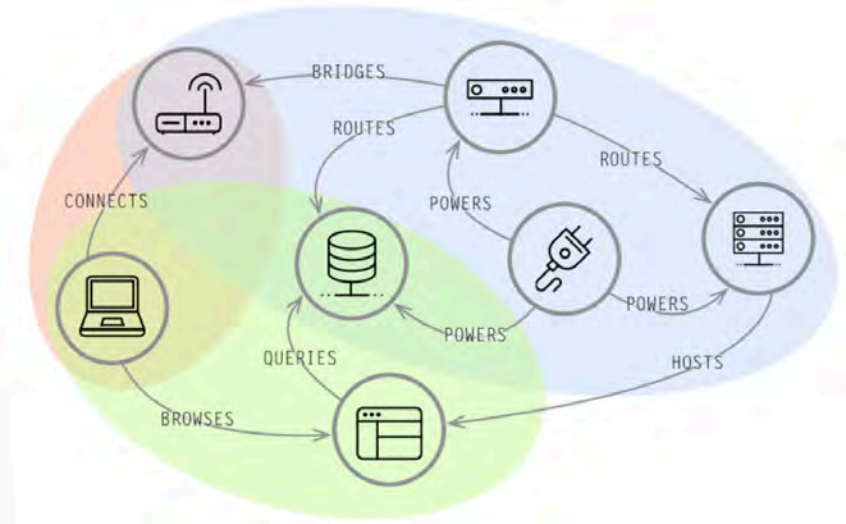


Figura 73: Modelado de redes en Neo4j[13]

2. **Social:** como se ha comentado anteriormente, las redes sociales son uno de los casos de uso en los que mejor se adapta Neo4j. Familiares, amigos y seguidores se extienden en un grafo social que revela patrones de comportamiento similar.

[11] http://neo4j.com/api_docs/1.2/org/neo4j/kernel/HighlyAvailableGraphDatabase.html
[12] http://neo4j.com/docs/stable/rest-api.html
[13] http://neo4j.com/use-cases/

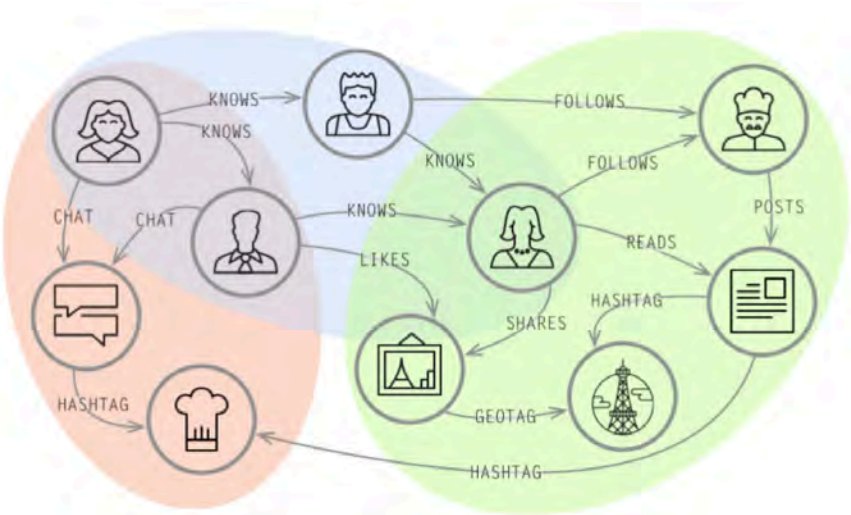


Figura 74: Modelado de una red social en Neo4j[13]

3. **Recomendaciones:** al igual que con las redes sociales, Neo4j ofrece una gran facilidad para modelar problemas relacionados con los sistemas de recomendación. Conecta los puntos de interés, aparentemente sin relación, para efectuar recomendaciones basadas en familiares o amigos.

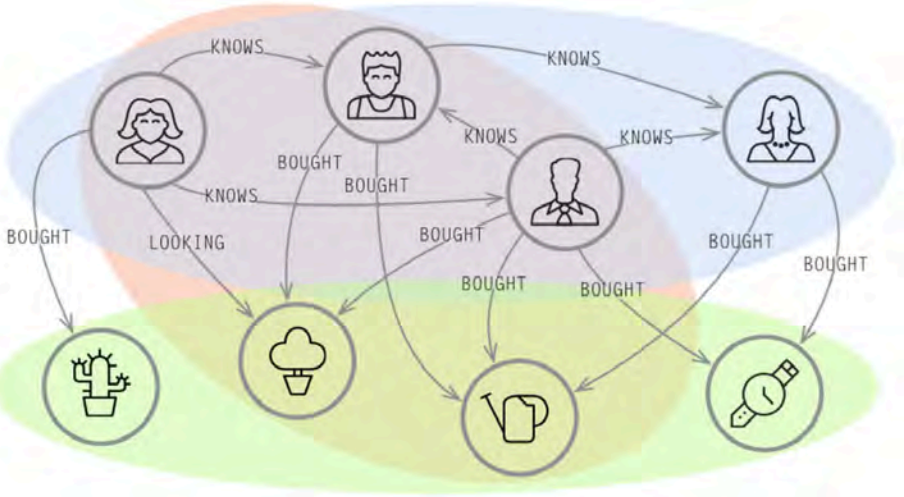


Figura 75: Modelado de un sistema de recomendaciones en Neo4j[13]

4. **Identificación y gestión del acceso:** quién eres y qué acciones tienes permitidas dependiendo de las relaciones entre una organización, un sistema y tú mismo.

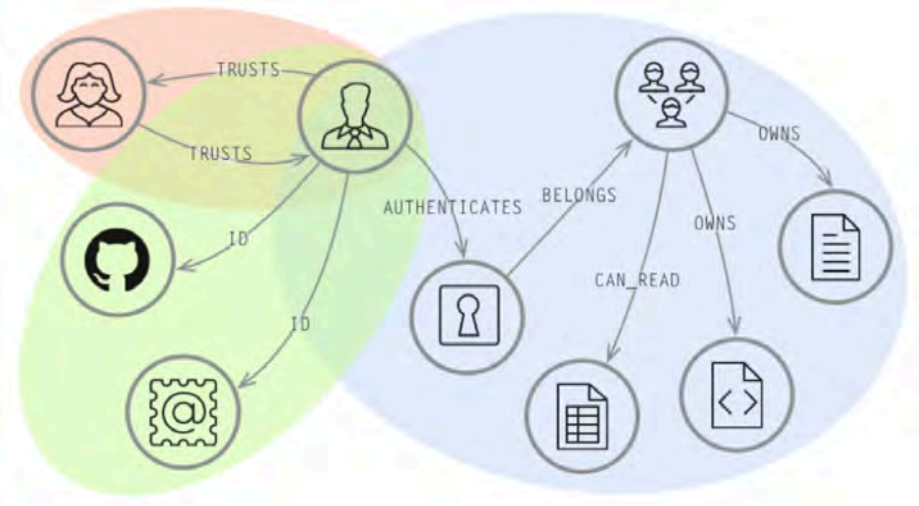


Figura 76: Modelado de un sistema de control de acceso en Neo4j[13]

5. **Gestión de datos maestros:** la organización y gestión de las personas dentro de una empresa, también se modela de manera sencilla en grafos. Jerarquías profundas de arriba hacia abajo, lateral y conexiones diagonales.

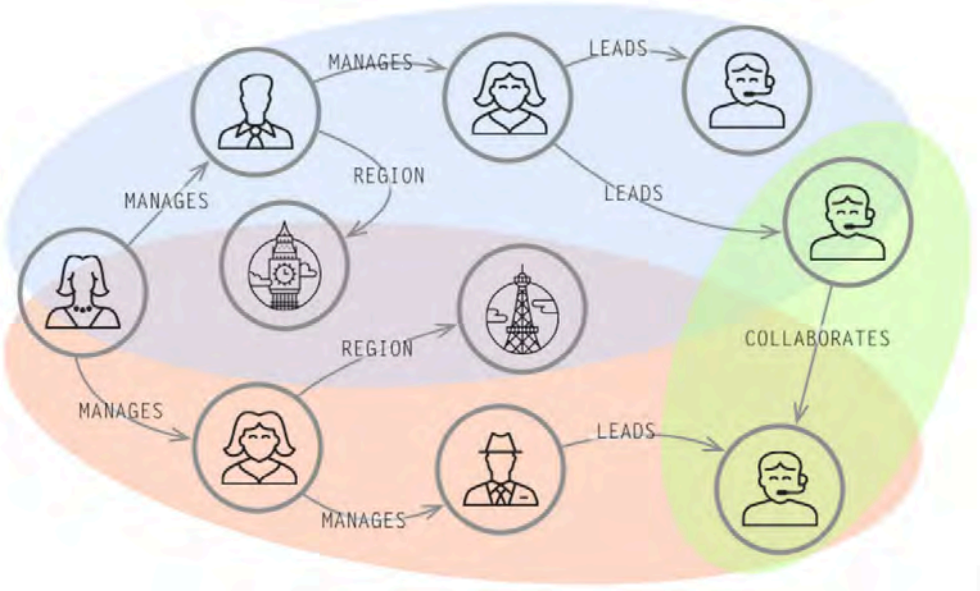


Figura 77: Modelado sobre la gestión de personas en Neo4j[13]

6. **Gestión de activos digitales:** la biblioteca multimedia nace de las relaciones entre los activos digitales y sus atributos.

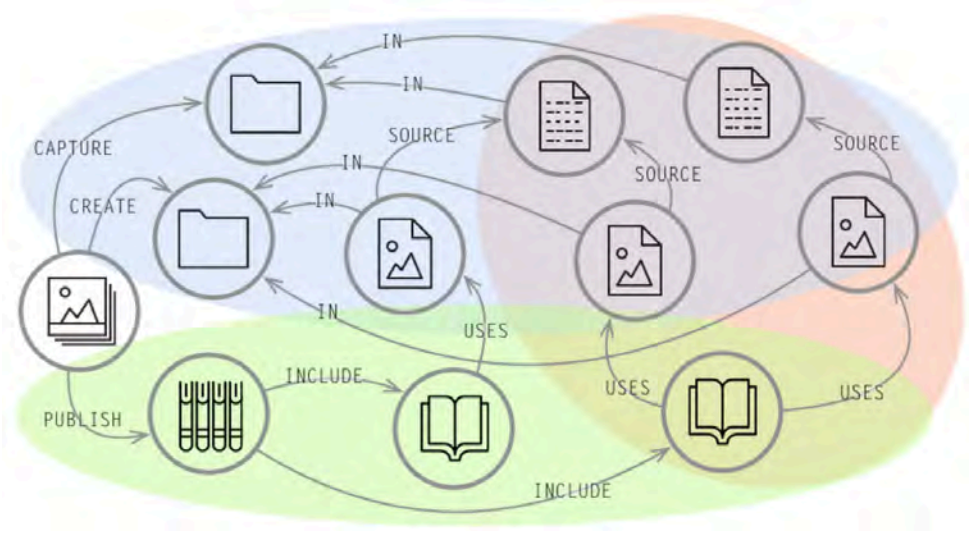


Figura 78: Modelado de la biblioteca multimedia en Neo4j[13]

7. **Detección de fraude:** análisis en tiempo real con relación de datos descubre anillos de fraude y otras estafas sofisticadas.

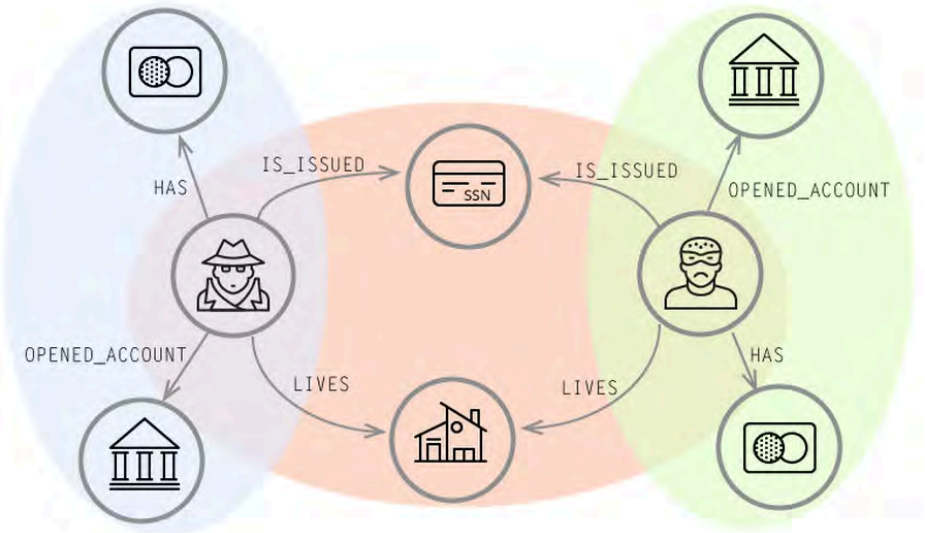


Figura 79: Modelado para detección de fraude en Neo4j[13]

2.3.3.2 KNOWLEDGE MANAGER[14]

KnowledgeMANAGER (KM) es una herramienta cuyo propósito fundamental es la gestión de ontologías. Ha sido creada por la empresa europea de tecnologías de la información conocida como "The Reuse Company". El desarrollo de esta herramienta se ha realizado (y se realiza actualmente el mantenimiento) por miembros de la Universidad Carlos III de Madrid (campus de Leganés), más concretamente por el Departamento de Informática. KM reúne un conjunto de características entre las que se encuentran:

[14] http://www.reusecompany.com/knowledgemanager

- Cuestiones lingüísticas
- Gestión del vocabulario
- Relaciones
- Patrones
- Tesauros

Hoy en día, el conocimiento es el activo más valioso para las organizaciones modernas. Una gestión adecuada de la organización del conocimiento es la clave para tener éxito. El conocimiento debe ser obtenido de numerosas fuentes y almacenado en repositorios seguros. De esta forma, se convertirá en un componente reutilizable para los proyectos software.

KM está enfocado precisamente hacia la gestión del conocimiento. Permite crear conocimiento para ser reutilizado posteriormente apoyándose en distintos conceptos como la creación de vocabularios, tokenización y normalización de términos, el uso de etiquetas sintácticas, desambiguación de términos y el uso de patrones. KM permite crear patrones a través de los cuáles se indexan términos, creando artefactos compuestos por distintos elementos de conocimiento, formando así una red de términos (marcados con etiquetas sintácticas) y relaciones entre ellos, que consigue generar conocimiento.

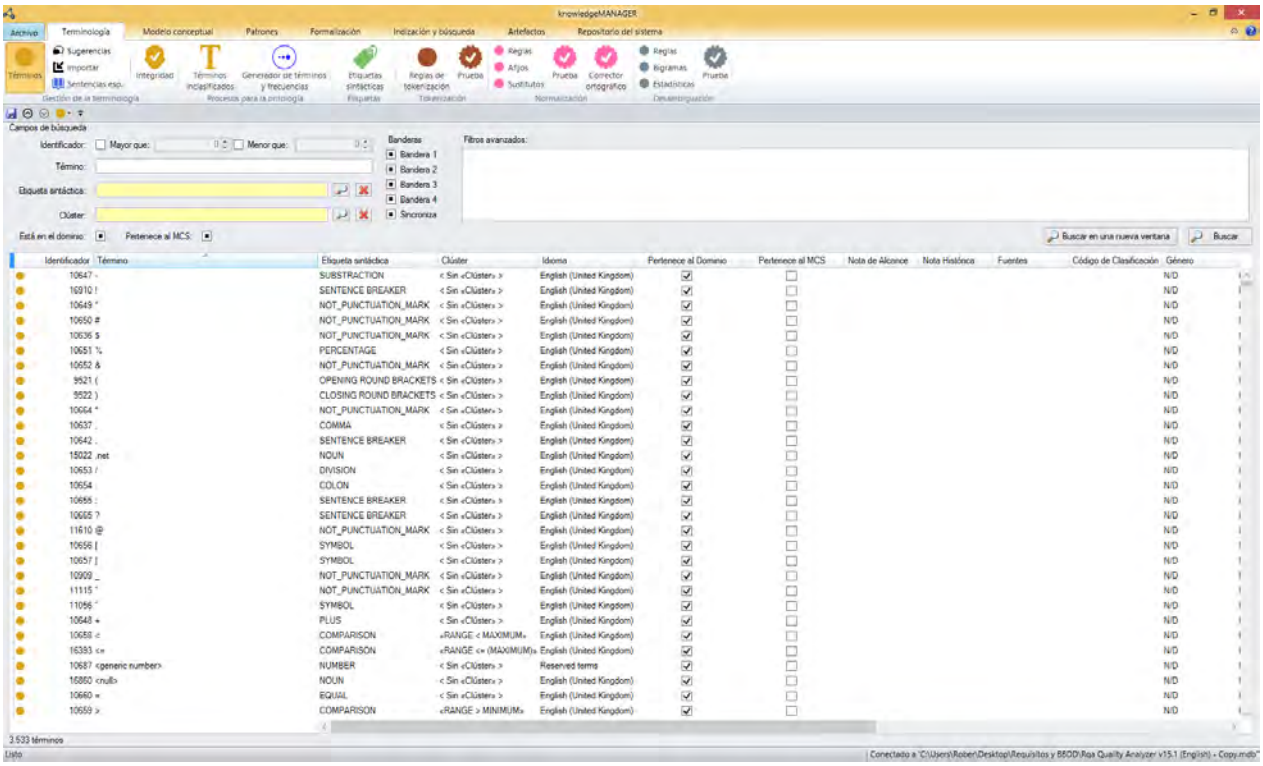


Figura 80: Herramienta KnowledgeMANAGER

En la *Figura 80*, se observan las distintas pestañas que dispone KM. Para el presente estudio, las pestañas que se han utilizado han sido, principalmente, las de *Indización y búsqueda* y *Artefactos*. En el capítulo *Experimentación*, se detallan cada una de las pantallas de esta herramienta, que han permitido indexar los datos y ejecutar las consultas.

2.3.3.2.1 Modelo RSHP

KM se incluye en esta sección de Bases de Datos Orientadas a Grafos ya que, de forma implícita, crea un grafo mediante el **modelo RSHP**. Este modelo RSHP junto con Neo4j, son los dos elementos fundamentales del presente estudio. En el capítulo 6 de este documento, se incluye la comparativa que se ha realizado entre ambos, en lo que se refiere a cómo de buena es la extracción de información en cada uno de ellos.

El **modelo de representación del conocimiento universal, RSHP**, se basa en la idea de que **cualquier información puede ser descrita por un conjunto de relaciones entre diferentes conceptos**. Con este planteamiento, el elemento principal de una unidad de información es la relación. Por ejemplo, los modelos de datos entidad-relación están representados como relaciones entre distintos tipos de entidad, o los modelos de objetos software, que pueden ser representados mediante relaciones entre objetos o clases.

RSHP incluye un modelo de repositorio para almacenar información y relaciones con el fin de reutilizar todo tipo de conocimiento. Además, el lenguaje natural puede ser representado también por relaciones entre distintos términos, utilizando la misma estructura que en los ejemplos anteriores. En concreto, para representar el lenguaje natural, se deben utilizar oraciones bajo la estructura Sujeto + Verbo + Predicado (SVP), la cual puede ser considerada como una relación V entre el sujeto S y el predicado P. Principalmente, RSHP se basa en los siguientes principios:

- **El elemento principal de descripción es la relación**, ya que es el encargado de vincular elementos de conocimiento.
- Un elemento de conocimiento, en inglés **Knowledge Element (KE)**, es un **componente atómico de conocimiento** que aparece en un artefacto y que está vinculado, por medio de una o varias relaciones, con otros KEs para construir información. Se define por un concepto, el cual es representado por un término normalizado (palabra clave de un vocabulario o dominio controlado). Por su parte, los artefactos son contenedores de conocimiento de KEs y sus relaciones.

En RSHP, el modelo de representación para describir el contenido de cualquier tipo de artefacto (requisitos, riesgos, modelos, pruebas, documentos de texto o código fuente), se debe realizar por medio de RSHPs, donde cada RSHP es denominada “RSHP-descripción” y debe describirse mediante KEs. Una característica importante de este modelo es que no hay ninguna restricción para representar un tipo de conocimiento en particular. Además, RSHP ha sido utilizado como modelo de información para construir sistemas de indexación y recuperación de propósito general, modelos de representación de dominio, para la evaluación de la calidad de requisitos y para herramientas de gestión del conocimiento como KnowledgeMANAGER.

Para entender mejor este modelo, a continuación se incluye un diagrama en el que se representan sus principales elementos:

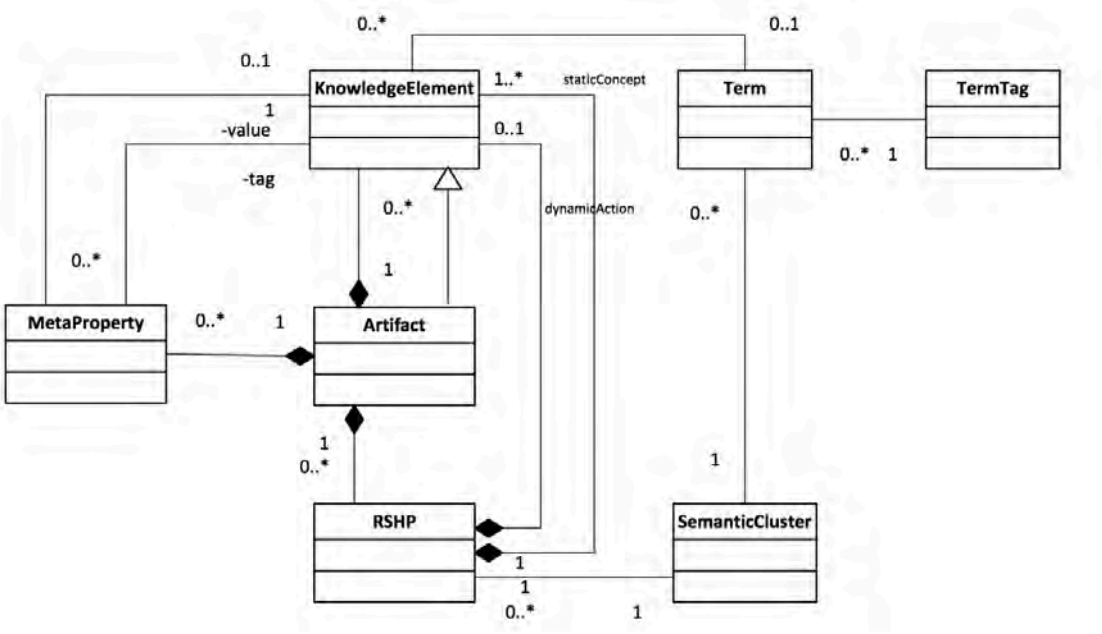


Figura 81: Modelo RSHP

A partir de la *Figura 81*, se comprueba como el modelo RSHP está compuesto por los siguientes elementos:

- **Artifact:** como se ha comentado, es un contenedor de relaciones y elementos de conocimiento (KnowledgeElements).
- **RSHP:** es una relación entre distintos elementos de conocimiento.
- **KnowledgeElement (KE):** es un elemento de conocimiento. También puede identificarse como la aparición de un término.
- **Term:** es un string o literal.
- **TermTag:** representa la categoría sintáctica de cada término.
- **MetaProperty:** representa las propiedades que posee cada KE.
- **SemanticCluster:** interviene en la creación de relaciones RSHP entre distintos términos.

Por ejemplo, si se tuviese la oración "El coche tiene freno", se tendrían los términos "El", "coche", "tiene" y "freno", cada uno de ellos con su categoría sintáctica, siendo determinante, sustantivo, verbo y sustantivo, respectivamente. Los términos "coche" y "freno" serían KEs y se crearía una relación RSHP entre ambos, a través del término "tiene". La RSHP resultante quedaría de la siguiente forma: RSHP-frenar = {KE1, KE2, "tiene"}, donde KE1 representa el término "coche" y KE2 el término "freno". Como se puede ver, el término "El" se obvia, ya que no aporta nada en la relación.

Como para la realización del experimento entre RSHP y Neo4j se han utilizado requisitos, un ejemplo de artefacto podría ser un requisito. En el apartado *Experimentación* se detallará todo el proceso de indexación de los requisitos en KnowledgeMANAGER, indicando cómo se crean los artefactos.

2.3.3.3 ALLEGROGRAPH[15]

AllegroGraph es una base de datos en grafo moderna, de alto rendimiento y persistente. Realiza una utilización eficiente de la memoria en combinación con el almacenamiento basado en disco, lo que le permite escalar a miles de millones de tripletas, manteniendo un rendimiento superior [15].

AllegroGraph trabaja con diferentes lenguajes y entornos de programación como por ejemplo Java, Python, http, JavaScript y Lisp. Además, como ocurría en Neo4j, las transacciones en AllegroGraph son ACID.

AllegroGraph es una base de datos y un marco de aplicación (framework) para construir aplicaciones de la web semántica. Es capaz de almacenar datos y metadatos como triples o tripletas. Podemos consultar estas tripletas a través de diferentes APIs de consulta como por ejemplo SPARQL y Prolog. Asimismo, permite aplicar razonamiento RDFS++ con el sistema inteligente que incorpora. AllegroGraph incluye soporte para análisis de redes sociales, capacidades geoespaciales y razonamiento temporal [16].

A continuación, se va a conocer de forma más concreta los tipos de datos que AllegroGraph permite almacenar. Para ello, se incluye el siguiente esquema, en el que se observa cierta información sobre Jans y sus mascotas.

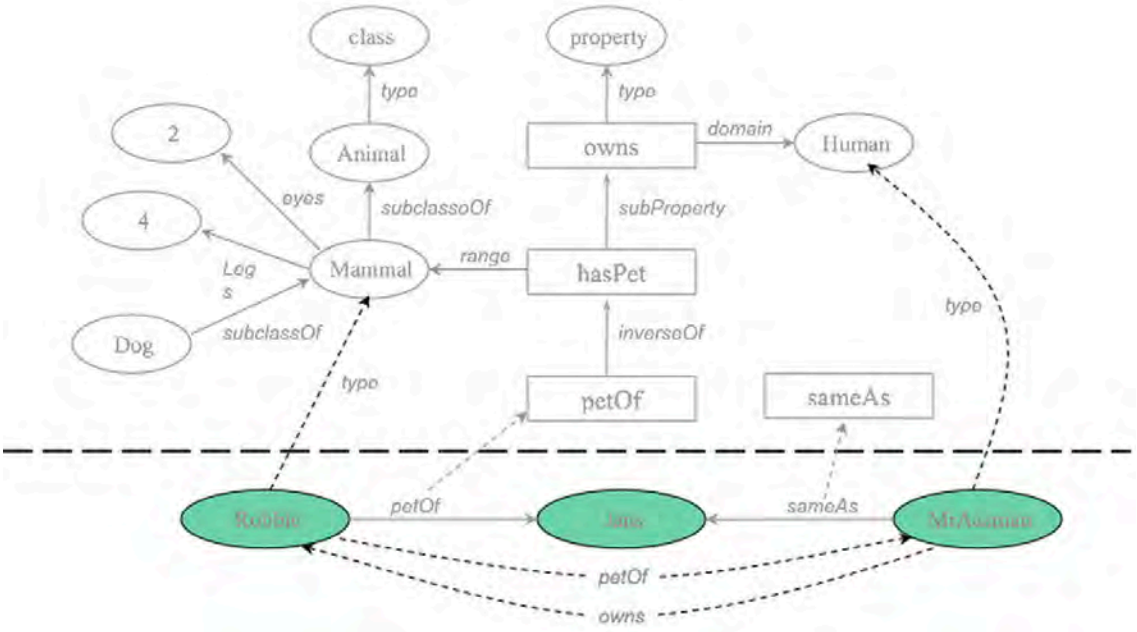


Figura 82: Esquema representando información (AllegroGraph)[16]

Como se comentaba anteriormente, en la *Figura 82* se puede observar cierta información sobre Jans y sus mascotas. Como ocurre en numerosos datos en la web, se pueden ver relaciones explícitas como por ejemplo *Robbie es la mascota de (petOf) Jans* y otras relaciones más de sentido común como por ejemplo *petOf es una relación inversa de (inverseOf) hasPet* o *perro (dog) es una subclase de (subclassOf) mamífero (Mammal)*.

[15] http://allegrograph.com/
[16] http://franz.com/agraph/support/documentation/current/agraph-introduction.html

Existen numerosas formas de almacenar esta información, pero como se veía en el capítulo *"LENGUAJES DE RECUPERACIÓN DE LA INFORMACIÓN - SPARQL"*, la **W3C** ha estandarizado esto mediante **RDF**. En el caso de que se tuviesen muchas tripletas de diferentes contextos, se podría añadir un nuevo campo a la afirmación llamado *named graph* que en español vendría a ser como el nombre del grafo al que pertenece la tripleta. Aunque ahora las afirmaciones consten de 4 campos, se seguirán llamando tripletas.

Sujeto	Predicado	Objeto	Grafo
Jans	Type	Human	Página de inicio de Jans
Robbie	petOf	Jans	Página de inicio de Jans
petOf	inverseOf	hasPet	Gramática inglesa
Dog	subClassOf	Mammal	Ciencia

Tabla 1: Extracción de tripletas en AllegroGraph[16]

En la *Tabla 1* están representadas como tripletas las afirmaciones que se analizaban más arriba. La visión de la web semántica busca que las páginas web contengan suficientes datos auto-descriptivos para que las máquinas puedan navegar por ellas como lo hacen los humanos. Esto permitirá a los equipos dar mejores respuestas a las preguntas de los usuarios. AllegroGraph es una base de alto rendimiento diseñada para almacenar este tipo de información, consultarla y razonar con ella.

Un aspecto importante que se debe resaltar y por el cual se ha incluido AllegroGraph en este trabajo, es que no restringe el contenido de sus tripletas a RDF puro. Es decir, mediante AllegroGraph también se puede representar esa información de la que se ha estado hablando en grafos. De esta forma, se representarían en nodos tanto el sujeto como el objeto de la tripleta y en una relación entre ambos nodos se incluiría el predicado, creando una tripleta por cada relación existente. El campo que se citaba anteriormente *named graph,* se podría utilizar para incluir información adicional específica de la aplicación. Utilizado de esta forma, AllegroGraph se convierte en una potente base de datos orientada a grafos.

2.3.3.4 SPARKSEE[17]

Sparksee (anteriormente conocido como DEX) es una base de datos orientada a grafos escrita en C++ que permite realizar rápidos análisis de grandes redes, por lo que se caracteriza por un alto rendimiento [17]. Se encuentra disponible de forma nativa para los lenguajes .Net, C++, Python, Objective-C y Java. Además está disponible para todos los sistemas operativos. Con su versión para móviles, Sparksee se ha convertido en la primera base de datos orientada a grafos disponible para iOS y Android.

Sparksee, al estar basado en bases de datos orientadas a grafos, se caracteriza por tener los datos estructurados en grafos, realizar las consultas con operaciones orientadas a grafos y tener restricciones para garantizar la integridad de los datos.

Sparksee está diseñado para **multi-grafos** a gran escala **dirigidos**, **etiquetados** y con **atributos**. Es decir, cada nodo y relación tendrán asignado un tipo de etiqueta y podrán tener los atributos que se deseen en cada uno de ellos (no tienen porqué ser los mismos), se permiten relaciones tanto dirigidas como no dirigidas y entre dos nodos podrán existir múltiples relaciones. Está basado en colecciones de identificadores de objetos almacenados como mapas de bits [18].

[17] http://www.sparsity-technologies.com/

Las principales características de Sparksee son las siguientes:

- El grafo se puede dividir en pequeñas estructuras para traer a memoria principal sólo las partes significativas (**caching**).
- **Identificadores de objeto** (oids) en lugar de objetos complejos.
- Estructuras específicas para **mejorar los recorridos** por el grafo: se indexan las relaciones y los vecinos de cada nodo.
- **Índices de atributos**: mejoran las consultas en las que se filtra por un valor de algún atributo.

Por otro lado, las capacidades que asegura Sparksee son las que se indican a continuación:

- **Eficiencia**: representación muy compacta utilizando mapas de bits.
- **Capacidad**: permite almacenar más de 100 mil millones de nodos y relaciones en un único equipo multi-núcleo.
- **Actuación**: respuestas por debajo del segundo en consultas de recomendación.
- **Escalabilidad**: alto rendimiento para consultas simultáneas.
- **Consistencia**: soporte transaccional parcial con recuperación.

En relación con la representación en mapas de bits, cabe destacar que cada nodo y relación es identificado por un único e inmutable oid. Cada conjunto de nodos o relaciones se representan en una estructura de mapa de bits. Cada posición en un mapa de bits corresponde con el oid de un objeto. Estas técnicas de compresión permiten reducir el espacio de almacenamiento. Las operaciones lógicas binarias son muy eficientes con estas estructuras.

Con todos estos datos, una **representación de un conjunto de valores** se efectuaría por medio de **dos mapas**, como se puede observar en la siguiente figura:

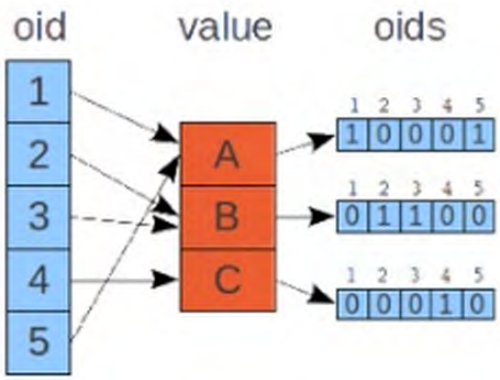


Figura 83: Representación de un conjunto de valores en Sparksee[18]

Como se puede comprobar en la *Figura 83*, se necesitan dos mapas los cuales se detallan a continuación:

- EL mapa de la izquierda (oid) asigna un identificador oid a cada elemento de un conjunto de nodos o relaciones.

[18] http://es.slideshare.net/SparsityTechnologies/sparksee-technology-overview

- El mapa de la derecha (oids) asigna al oid de cada nodo o relación un valor binario, siendo 1 en el caso de que el valor representado sí esté siendo apuntado por el correspondiente oid del nodo o relación en cuestión, o 0 en caso contrario.

Seguidamente, se va a continuar repasando de forma superficial las distintas operaciones que se pueden realizar sobre un conjunto de valores en Sparksee.

- **Domain**: devuelve el conjunto de diferentes valores.
- **Objects**: devuelve el conjunto de nodos o relaciones asociados a un valor.
- **Lookup**: devuelve el valor asignado a cada uno de los elementos de un conjunto de objetos.
- **Insert**: añade un nodo o relación a la colección de objetos de un valor.
- **Remove**: elimina un nodo o relación de la colección de objetos de un valor.

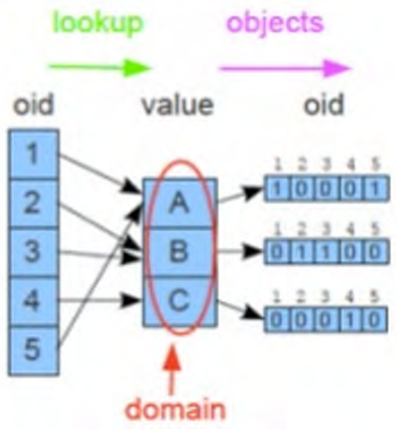


Figura 84: Operaciones sobre un conjunto de valores en Sparksee[18]

2.3.3.4.1 ¿Por qué Sparksee?

Existen cuatro puntos fundamentales por los que es aconsejable utilizar una base de datos basada en grafos de Sparksee. A continuación se detalla cada uno de ellos:

1. **Pequeños espacios de memoria fáciles de manejar**: el grafo es representado por estructuras de datos de mapas de bits que permiten alcanzar altas tasas de compresión. Adyacencias de los nodos son representadas por mapas de bits para minimizar su espacio. Cada valor en la base de datos está representado una única vez, evitando así la replicación innecesaria.
2. **Eficiencia de la Entrada/Salida (E/S) recuperando datos de la base de datos**: cada uno de los mapas de bits se divide en trozos que encajan en páginas de disco, lo cual permite mejorar la localización en la E/S. Además, el número de veces que cada página de datos es traída a memoria se minimiza aplicando avanzadas políticas de E/S.
3. **Impulsar el rendimiento al máximo:** usando mapas de bits, las operaciones se calculan mediante instrucciones lógicas binarias que simplifican la ejecución. El núcleo C++ evita una ejecución excesiva y la gestión compleja de la memoria, a diferencia de los motores basados en Java. Alto rendimiento en condiciones de estrés.

4. **Orientada al usuario**: indexación completamente nativa que permite un acceso extremadamente rápido a cada una de las estructuras de datos del grafo. Disponible de forma nativa en .Net, C++, Python, Objective-C y Java además de iOS y Android. Disponible API de bajo nivel con acceso directo a las funcionalidades principales del motor.

2.3.3.4.2 Casos de Uso

En este apartado se van a indicar los principales casos de uso en los que es idóneo utilizar Sparksee. Seguidamente se cita a cada uno de ellos:

- **Análisis de redes sociales**: realiza operaciones de análisis de forma rápida de Twitter o Facebook para las aplicaciones de los clientes.
- **Redes bibliográficas**: implementa herramientas analíticas para extraer y relacionar información de redes tales como Wikipedia o IMDB.
- **Análisis de los medios de comunicación**: útil para recomendar nuevos contenidos y relacionar los datos sociales con los datos multimedia.
- **Seguridad de las redes y detección del fraude**: clave en análisis de patrones o registro de seguridad.
- **Exploración y optimización de redes físicas**: optimiza rutas logísticas, identifica cuellos de botella en la red y busca la distancia más corta para llegar a un determinado lugar.
- **Redes biológicas**: analiza y expresa en un grafo, redes genéticas y de proteínas. Detecta mutaciones y la evolución en el tiempo.
- **Recomendaciones en el comercio electrónico**: permite obtener y almacenar información como por ejemplo "¿Qué personas han comprado este artículo también?".

2.3.3.5 GRAPHBASE[19]

GraphBase es un sistema de gestión de bases de datos basadas en grafos (DBMS) de segunda generación [19]. GraphBase está diseñado para manejar estructuras de datos grandes y complejas. Este sistema gestor de BDOG hace posible construir almacenes de datos masivos y altamente estructurados, ya que fue construido para gestionar grandes grafos. El poder de GraphBase proviene de su rendimiento, su sencillez, su versatilidad y sus herramientas únicas. Seguidamente, se va a detallar cada una de estas cualidades.

2.3.3.5.1 Características Generales

Rendimiento

GraphBase ha sido construido para servidores multiprocesador modernos y está diseñado para obtener el máximo beneficio de RAMs de gran tamaño y del almacenamiento de alta velocidad [20]. Un sólo servidor de bajo coste de GraphBase puede manejar miles de millones de consultas y actualizar diariamente mil millones de nodos y mil millones de relaciones.

[19] http://graphbase.net/

El verdadero secreto para conseguir un rendimiento tan alto reside en la sofisticada gestión de hilos y las estructuras compactas utilizadas, lo cual permite que resida en memoria gran parte del grafo. GraphBase también incluye algunas innovaciones únicas como por ejemplo, las heurísticas de las relaciones, que permiten realizar recorridos por el grafo y una velocidad de consulta de 10 a 100 veces más rápida que en otras implementaciones de BDOG.

Existen ciertas situaciones en las que es preferible guardar el grafo en varios servidores en lugar de sólo en uno. A veces no es posible mantener el grafo entero en un sólo servidor y en otras ocasiones, es preferible distribuir el grafo para que el procesamiento pueda ser también distribuido. GraphBase está diseñado para ser distribuido basándose en el estilo de la "Nube" o "Cloud" en inglés. Cada servidor es autónomo, la comunicación entre ellos es asíncrona y las estrategias sofisticadas de almacenamiento en caché y colas permiten, a un conjunto de servidores GraphBase, controlar las cuestiones de latencia y ancho de banda de una nube distribuida geográficamente. Las relaciones en GraphBase están encapsuladas dentro de cada nodo, lo cual simplifica en gran medida la distribución de los datos.

GraphBase permite incorporar almacenes de datos simples, comprimidos, altamente eficientes y enfocados a los nodos. Es una estrategia tan eficaz que permite analizar grandes cantidades de datos en tiempo real (Big Data).

Sencillez

GraphBase simplifica de forma notoria el trabajo almacenando los datos en grafos estructurados. Es una base de datos que permite pensar en grafos, usando grafos. Además, incluye su propia forma de realizar consultas sobre grafos, mediante el lenguaje denominado "**Bounds Language**". En este lenguaje se pueden ejecutar consultas sencillas en las que se especifican unos límites para las mismas, indicando las interacciones entre recorridos simultáneos a través del grafo. Los resultados de las consultas son en forma de grafo.

Es importante destacar que este sistema incluye un **API Java**[20], a través del cual se pueden realizar numerosas acciones, como por ejemplo transformar un grafo en objetos con una sola línea de código.

Versatilidad

Hoy en día, no hay certeza de cuál es la estructura ideal de un grafo para almacenar datos. Almacenamiento en tripletas ("Triplestore"), Grafo de Propiedad, HyperGraph entre otros, cada uno de ellos tiene sus ventajas y sus inconvenientes. La elección suele depender de la naturaleza de los datos y del propósito para el que se vayan a utilizar [21].

GraphBase Enterprise Edition[21] permite configurar primitivas de peso ligero para obtener la estructura que cada cliente desee. También permite indicar cómo se indexa el grafo. Asimismo, mediante la versión GraphBase Agility Edition[22], da la posibilidad al usuario de seleccionar qué partes del grafo necesitan poder ser bloqueadas y cuáles no, sobre qué partes del grafo se puede hacer rollback y sobre cuáles no, además de otras funcionalidades de soporte transaccional.

[20] http://graphbase.net/JavaAPIHelp.html
[21] http://graphbase.net/Enterprise.html
[22] http://graphbase.net/Agility.html

2.3.3.5.2 Herramientas de GraphBase

GraphBase simplifica la gestión de los datos estructurados en grafos a través de sus herramientas enfocadas al grafo. Es importante destacar que en GraphBase la unidad de trabajo es el grafo, no el nodo. A continuación se indican las funcionalidades básicas que ofrece este sistema:

- **Añadir un grafo** a la base de datos.
- **Consultar la base de datos mediante "bounds queries"**, consultas en las que se indican unos límites para obtener un sub-grafo del grafo que se está preguntando.
- **Convertir el grafo en** una colección de **POJOs** (Plain Old Java Object) en Java en una línea de código.

A continuación se citan los diferentes productos que ofrece GraphBase:

- **GraphBase Agility Edition**: es la edición básica de GraphBase.
- **RapidGrapher For Agility**: permite transformar los datos de una base relacional a un grafo sencillo, en el que se puede buscar y navegar.
- **ViewServer For Agility**: es un servidor web a medida que permite navegar por los datos de forma sencilla.
- **RDF+ For Agility**: permite que GraphBase sea utilizado para aplicaciones de la Web Semántica.
- **GraphBase Enterprise Edition**: herramienta para resolver grandes y complejos problemas de datos.
- **GraphBase I6**: aplicación de análisis de datos e inteligencia. Permite realizar análisis de flujos de datos entrantes en tiempo real además de obtener la inteligencia de red que los humanos o agentes automatizados pueden usar, antes de que sea demasiado tarde.

2.3.3.6 GRAPH ENGINE[23]

Graph Engine (GE), anteriormente denominado Trinity, es tanto un almacén de memoria RAM como un motor de computación elaborado por Microsoft Research. Como almacén de memoria RAM distribuida, GE organiza la memoria principal de un conjunto de máquinas como un espacio de direcciones global direccionable, para almacenar conjuntos de datos a gran escala. A través del almacén de RAM, GE permite un rápido acceso aleatorio a datos sobre un gran conjunto de datos distribuidos. Como motor de computación, GE ofrece APIs personalizadas por el usuario para implementar lógica de procesamiento en grafo [22].

[23] http://www.graphengine.io/

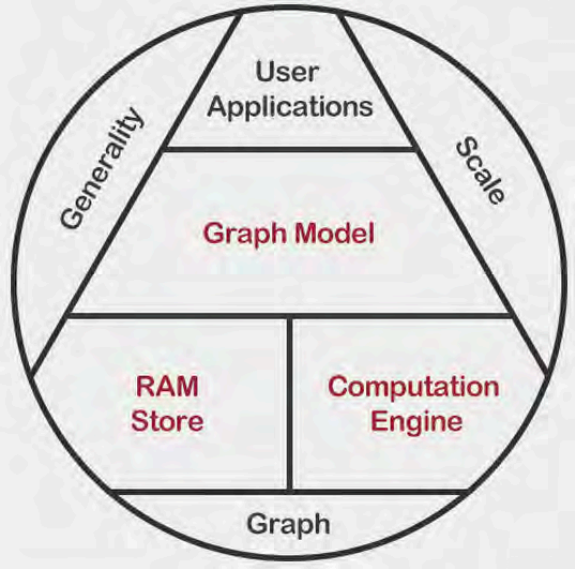


Figura 85: Esquema de Graph Engine[23]

La capacidad de explorar rápidamente datos junto con la computación paralela distribuida hacen de GE una gran plataforma de procesamiento de grafos. GE admite procesamiento de consultas online y proporciona un alto rendimiento en el análisis offline en miles de millones de nodos sobre grandes grafos.

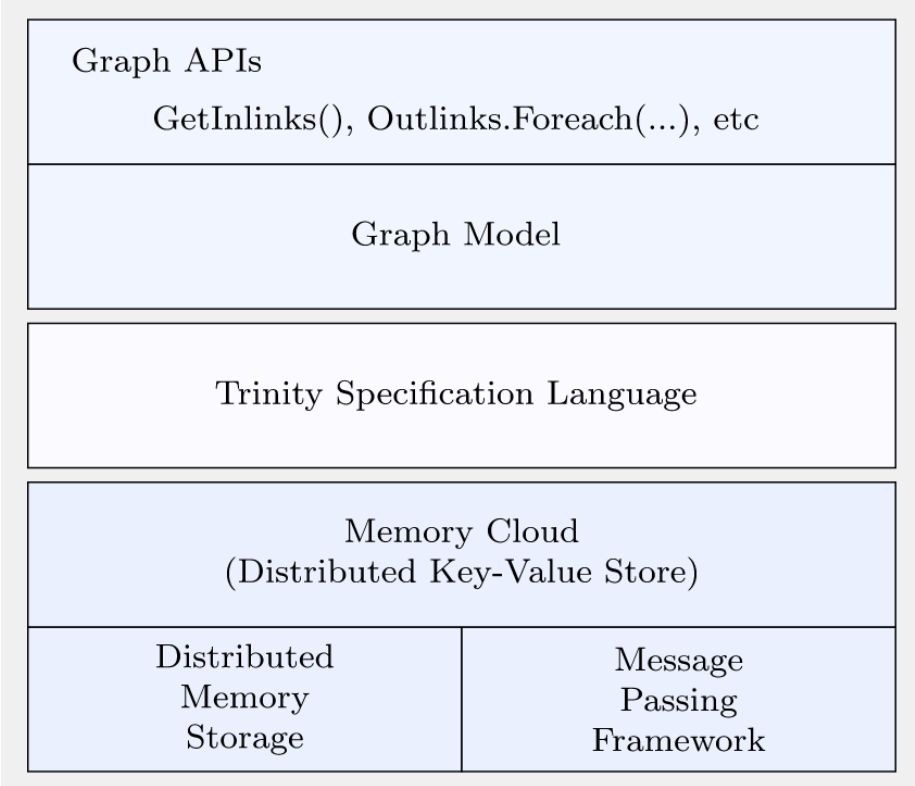


Figura 86: Pila de capas de Graph Engine[24]

[24] http://www.graphengine.io/docs/manual/

En la *Figura 86* se observa la estructura de capas que conforman Graph Engine. La "nube de memoria" o "*Memory Cloud*" es un almacén distribuido de pares clave-valor el cual está respaldado por un módulo de almacenamiento de memoria (*Distributed Memory Storage*) y por un marco de paso de mensajes (*Message Passing Framework*). El módulo de almacenamiento de memoria gestiona la memoria principal de un conjunto de máquinas y proporciona mecanismos de control de concurrencia. El módulo de comunicación de red proporciona una infraestructura de paso de mensajes eficiente y máquina a máquina.

Además, GE provee una especificación de un lenguaje denominado Trinity Specification Language (TSL) que une el modelo de grafo con la infraestructura de almacenamiento y computación. En lugar de utilizar un esquema fijo de grafo y modelos fijos de computación, GE ofrece la posibilidad a los usuarios de usar TLS para especificar esquemas de grafo, protocolos de comunicación y paradigmas computacionales. Por último en la capa superior, se encuentran todos aquellos APIs para trabajar con grafos.

2.3.3.6.1 TSL

Trinity Specification Language (TSL) es un lenguaje declarativo diseñado para aportar flexibilidad al sistema. A través de TSL, GE permite a los usuarios crear libremente esquemas de datos y ampliar la capacidad del sistema mediante computación definida por el usuario del lado del servidor [23].

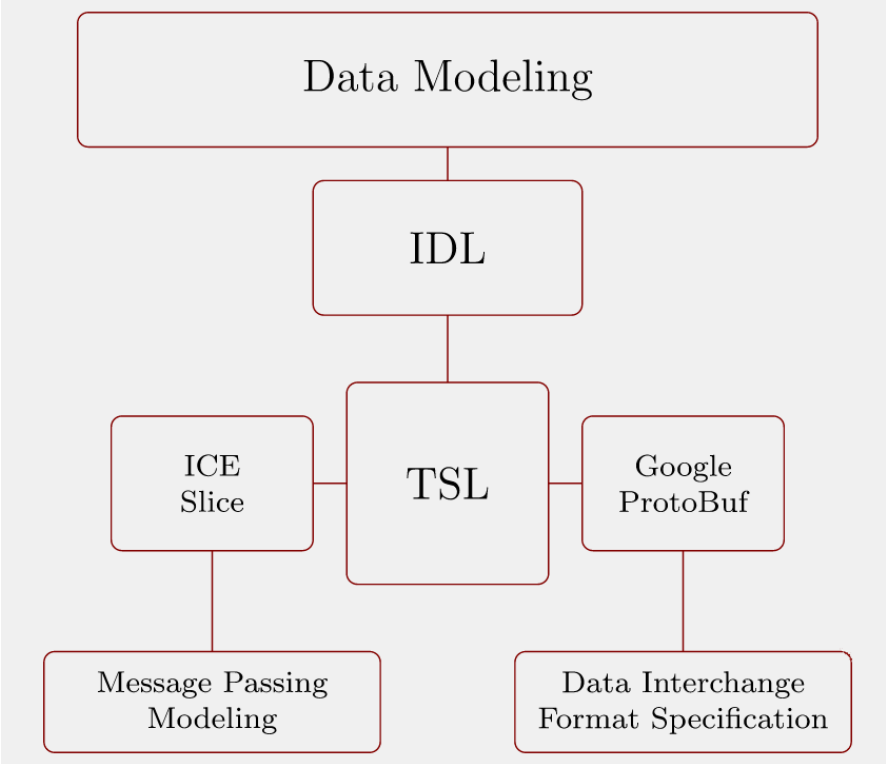


Figura 87: Trinity Specification Language (TSL) de Graph Engine[24]

A partir de la *Figura 87*, se comprueba que TSL aglutina principalmente las siguientes tres características: modelado de datos, modelado de protocolos de paso de mensajes y una especificación del formato del intercambio de datos. Este último punto permite a los usuarios

intercambiar datos, sin tener que serializar y deserializarlos, a través de una forma eficiente de codificar datos estructurados.

De la misma forma, TSL permite la manipulación de datos orientada a objetos, facilita la integración de los datos y facilita la ampliación del sistema. A continuación se muestra un pequeño ejemplo de modelado de datos con TSL [24].

```
[CellType: NodeCell]
cell struct Movie
{
    string Name;
    [EdgeType: SimpleEdge, ReferencedCell: Actor]
    List<long> Actors;
}
[CellType: NodeCell]
cell struct Actor
{
    string Name;
    [EdgeType: SimpleEdge, ReferencedCell: Movie]
    List<long> Movies;
}
```

Figura 88: Modelando un grafo de películas y actores con TSL[25]

Como se puede observar en la *Figura 88*, se definen dos tipos de nodos, uno denominado *Movie* y otro *Actor* usando dos estructuras de celda (*cell structs*). Este tipo de estructuras son elementos básicos a la hora de modelar grafos. Son contenedores de datos que pueden almacenar distintos tipos de datos como byte, int y double, además de distintas estructuras de datos como por ejemplo arrays, listas y demás estructuras definidas por el usuario.

En este caso, cada nodo contiene un atributo de tipo string denominado *Name* y una lista de tipo Long la cual actúa a modo de relación o arista con el otro nodo. Como se puede comprobar, antes de declarar la lista se incluye una línea en la que se indica, entre corchetes, el tipo de relación y el nodo al que se pretender referenciar. Además del tipo de relación *SimpleEdge*, también se aceptan relaciones del tipo *HyperEdge* y *StructEdge*.

2.3.3.7 INFINITE GRAPH[26]

Infinite Graph permite a las organizaciones lograr una mejor rentabilidad en sus datos de inversión, ayudándoles a conectar los puntos clave a escala global y a hacer preguntar más profundas y complejas sobre almacenes de datos nuevos o ya existentes [25]. Las principales ventajas que aporta Infinite Graph son las siguientes:

- **Escalabilidad**: altamente distribuido y solución escalable de BDOG.
- **Acceso simplificado**: acceso simplificado a los datos para su análisis mejorado y para el apoyo a las decisiones.
- **Rentabilidad**: bajo TCO debido a la baja necesidad de mover y transformar los datos.
- **Tecnología probada**: tecnología embebida en aplicaciones de misión crítica.

[25] http://research.microsoft.com/pubs/161291/trinity.pdf
[26] http://www.objectivity.com/products/infinitegraph/

2.3.3.7.1 Características Generales

En esta sección se van a detallar las principales características de Infinite Graph.

1. **Base de Datos Orientada a Grafos distribuida.**

Infinite Graph saca el máximo partido a los datos distribuidos utilizando lugares de almacenamiento configurables y flexibles. Cada usuario puede añadir sus lugares de almacenamiento a cada una de sus BDOG en cualquier momento, estando disponibles los datos inmediatamente para las aplicaciones que utilicen esa BDOG. Además, permite distribuir la carga de procesamiento de la forma más conveniente y eficiente según la aplicación.

2. **Intuitivo**. Incluye un API de Java enfocado a grafos y optimizado para las relaciones de datos.

3. **Colocación gestionada**.

Permite crear un modelo personalizado en el que colocar físicamente elementos del grafo, que son accedidos con más frecuencia, para mejorar el rendimiento de las consultas. Asimismo, se puede separar o aislar físicamente los objetos de acceso frecuente para evitar la contención de bloqueo.

4. **Transacciones ACID.**

5. **Consultas de navegación a gran alcance.**

Infinite Graph incluye un API de navegación con el que consultar el grafo, ejecutando "in-process" o en servidores remotos que distribuyen la carga de procesamiento y maximizan la eficiencia ejecutando las consultas donde se encuentran los datos.

6. **Potente filtrado.**

A través de un objeto de vista de grafo, se pueden utilizar distintas técnicas de filtrado para omitir objetos o tipos de objetos no relevantes para la consulta.

7. **Indexación y consulta.**

Infinite Graph permite crear índices automáticos a nivel de grafo sobre varios campos clave. Asimismo, ofrece la posibilidad de crear objetos de consulta reutilizables para realizar exploraciones, a nivel de grafo, de alto rendimiento. Los objetos de consulta aprovechan de forma automática los índices a nivel de grafo que sean relevantes, para lograr alcanzar un rendimiento óptimo.

8. **Visualización de datos.**

Además, Infinite Graph dispone de una herramienta flexible para visualizar los grafos que permite navegar por los datos y realizar consultas de navegación mediante la carga de "plugins" de navegación personalizados.

9. **Utilización del lenguaje de consulta PQL (Predicate Query Language).**

2.3.3.7.2 Casos de uso

Los principales sectores con los que Infinite Graph trabaja son los que se indican a continuación [26]:

1. **Gobierno**

Agencias de gobierno desarrollan un alto rendimiento, análisis altamente escalables e implementan soluciones de apoyo a la decisión. Estas soluciones de análisis de Big Data de carácter crítico, ofrece al gobierno de los EE.UU. la fiabilidad y precisión necesarias para mantener la excelencia a escala global o específica.

2. **Telecomunicaciones y redes**

En un mundo cada vez más interconectado, las telecomunicaciones y los proveedores de servicios de red deben mantenerse al día para estar siempre actualizados. Los clientes de Infinite Graph son capaces de desarrollar soluciones de alto rendimiento de forma eficiente, lo que asegura su excelencia y ventaja competitiva en los diferentes mercados en los que trabajan.

3. **Cuidados de la salud**

Ser capaz de acceder a datos críticos de un paciente en tiempo real puede ser la diferencia entre salvar una vida o no. Infinite Graph permite, a sus clientes del ámbito de la salud, desarrollar soluciones de tiempo real con las que analizar y gestionar datos de forma eficiente y precisa.

4. **Finanzas**

5. **Redes sociales**

Hoy en día, la gran mayoría de las personas se comunican por medio de las redes sociales. Infinite Graph permite a sus clientes desarrollar redes sociales, educacionales y comerciales, capaces de arrojar información valiosa, basada en las relaciones entre personas, a la hora de recomendar productos y experiencias.

2.3.3.8 HYPERGRAPHDB[27]

HyperGraphDB, principalmente, es un mecanismo de almacenamiento de datos de código abierto, basado en una potente aplicación de la gestión del conocimiento conocida como “directed hypergraphs” o “hipergrafos dirigidos”. Puede ser utilizado como una base de datos orientada a objetos embebida en proyectos java o como una BDOG [27].

[27] http://www.hypergraphdb.org/index

2.3.3.8.1 Características generales

Las principales características de HyperGraphDB son las que se indican a continuación:

- Potente modelado de datos y representación del conocimiento.
- Almacenamiento orientado a grafos.
- Relaciones N-arias entre los nodos del grafo.
- Recorridos a lo largo del grafo y consultas de tipo relacional.
- Indexación personalizable.
- Gestión del almacenamiento personalizable.
- Totalmente transaccional y multi-hilo.
- Transacciones ACID
- Marco P2P para la distribución de datos.

2.3.3.8.2 ¿Qué es un hipergrafo?

Un hipergrafo es una generalización de un grafo en el que cada relación puede conectar cualquier número de nodos. Formalmente, un hipergrafo H es un par H = (N, R), donde N es el conjunto de nodos que forman el grafo y R es el conjunto de relaciones del grafo expresadas como subconjuntos no vacíos de N. Seguidamente se muestra un ejemplo:

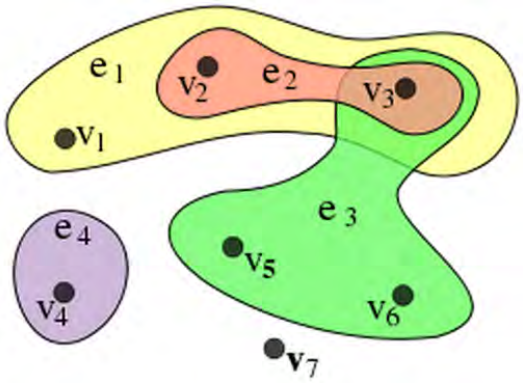


Figura 89: Ejemplo de hipergrafo[28]

La representación del hipergrafo de la *Figura 89* sería: H = (N, R) donde N = {v1, v2, v3, v4, v5, v6, v7} y R = {e1, e2, e3, e4} o lo que es lo mismo R = {{v1, v2, v3}, {v2, v3}, {v3, v5, v6}, {v4}}.

Como se comentaba anteriormente, HyperGraphDB está basado en hipergrafos dirigidos. Por lo que, sabiendo qué es un hipergrafo, un hipergrafo dirigido es aquel hipergrafo cuyas relaciones poseen dirección, como se puede observar en la siguiente figura [28]:

[28] https://es.wikipedia.org/wiki/Hipergrafo

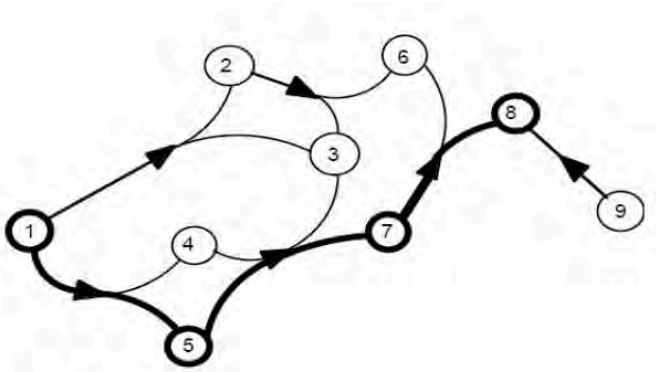


Figura 90: Ejemplo de hipergrafo dirigido[29]

A partir de la *Figura 90*, se comprueba cómo es la disposición de un hipergrafo dirigido, en la que una relación puede conectar varios nodos y cada una de las relaciones tiene dirección. Este es el modelo de almacenamiento que utiliza HyperGraphDB para gestionar el conocimiento.

2.3.3.8.3 Arquitectura de HyperGraphDB

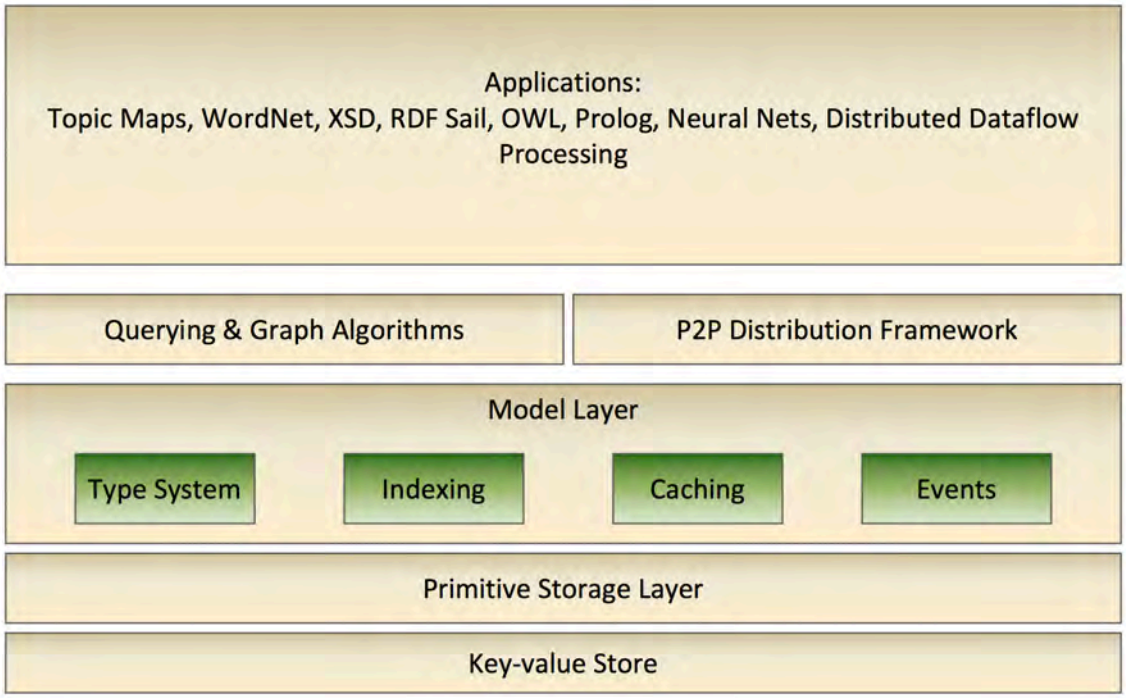


Figura 91: Arquitectura HyperGraphDB[29]

A partir de la *Figura 91* se comprueba como en la capa superior de la arquitectura de HyperGraphDB se encuentran las aplicaciones, en la capa inferior se encuentran los algoritmos de consulta y grafo y el *Framework Distribution P2P*. En el centro de la arquitectura se encuentra el *Model Layer*, el cual se encuentra por encima de las capas *Primitive Storage Layer* y *Key-value Store*.

[29] http://www.hypergraphdb.org/docs/HyperGraphDB-Presentation.pdf

La capa *Primitive Storage Layer* se corresponde con un grafo de bajo nivel de identidades y datos en bruto, mientras que la capa *Model Layer* se corresponde con un diseño para representar hipergrafos.

2.3.3.9 ORIENTDB[30]

OrientDB es una BDOG distribuida y multi-modelo de segunda generación. OrientDB es muy rápido ya que puede almacenar 220.000 registros por segundo en un hardware común. Incluso para bases de datos basadas en documentos, las relaciones se gestionan como en una BDOG, con conexiones directas entre los registros. Permite recorrer partes o incluso el grafo entero en milisegundos. Posee un fuerte sistema de perfiles de seguridad basado en usuarios y roles. Además, soporta SQL entre otros lenguajes de consulta. La capa SQL permite que OrientDB sea fácil de usar para los expertos de las bases de datos relacionales [29].

OrientDB se encuentra disponible en dos versiones: Community Edition, que es gratuita para cualquier uso y Enterprise Edition, software comercial de pago desarrollado por el mismo equipo que desarrolló el motor OrientDB.

2.3.3.9.1 Principales ventajas

Las principales ventajas que ofrece OrientDB son las que se detallan a continuación [30]:

1. **Construido para ser veloz.**

OrientDB ejecuta de forma rápida tanto lecturas como escrituras. En lo que se refiere a escrituras, puede almacenar hasta 400.000 registros por segundo. Además, permite insertar hasta 120.000 registros mientras se mantiene sincronizado el grafo para realizar análisis en tiempo real.

OrientDB es capaz de incrustar bases de datos basadas en documentos, pero también dispone soporte para las relaciones. Un punto importante a destacar es que no utiliza las costosas combinaciones JOIN. En su lugar, utiliza punteros persistentes y muy rápidos entre los registros, permitiendo recorrer partes de grafos o incluso grafos enteros en pocos milisegundos. La velocidad, a la hora de recorrer el grafo, se mantiene constante para cualquier tamaño de grafo.

2. **Máxima flexibilidad.**

OrientDB aglutina en un solo producto numerosas características entre las cuales se encuentran:

- Soporta conceptos orientados a objetos.
- Seguridad en usuarios y roles.
- Seguridad a nivel de registro.
- Bloqueo a nivel de registro.
- Admite SQL.

[30] http://orientdb.com/orientdb/

- Lenguaje de consulta basado en SQL.
- Transacciones ACID.
- Recorridos a través de relaciones.
- Tipos de datos personalizados.
- Escalabilidad elástica.
- Funciones del lado del servidor.
- HTTP Rest/JSON nativo.

3. Escalabilidad elástica

En OrientDB, el rendimiento no está limitado sólo por un servidor. El rendimiento global es la suma de los rendimientos de todos los servidores. Si el usuario pretende aumentar el rendimiento, sólo tiene que añadir un nodo de servidor en la red y este se unirá automáticamente al cluster de servidores distribuidos existente, sin tener que realizar ningún tipo de configuración. La sincronización se efectúa en el momento que el servidor se pone en línea.

4. Fiabilidad distribuida

Cuando un nodo de servidor se bloquea, OrientDB, gracias a WAL (Write Ahead Logging), es capaz de restaurar el contenido de la base de datos. Las transacciones pendientes se deshacen automáticamente. Además, el cluster de servidores redistribuye la carga entre lo nodos de servidor disponibles y todos los usuarios que estaban conectados al nodo bloqueado se cambian automáticamente a un nodo disponible.

5. Fácil de instalar y usar.

OrientDB está escrito totalmente en Java y puede ejecutarse en cualquier plataforma sin necesidad de instalaciones ni configuraciones. Si se utiliza otro lenguaje distinto de Java, OrientDB proporciona distintos drivers con los que se puede utilizar el lenguaje deseado.

6. Bajo TCO (Total Cost of Ownership o Coste Total de Propiedad).

Con el uso de OrientDB Community Edition no existe ningún tipo de coste asociado. OrientDB se distribuye con una sola licencia (Apache 2 Open Source), lo que permite su uso de forma gratuita para cualquier propósito. Además, al ser un sistema bastante completo, reduce la necesidad de tener que instalar distintos sistemas para lograr alcanzar los objetivos.

7. Listo para las empresas.

OrientDB Enterprise Edition, es una extensión de la versión gratuita y ofrece a las empresas funciones como: consulta de perfiles, configuración de clusters distribuidos, grabación de métricas y monitorización en vivo con alertas configurables.

8. Producto sólido, estable y maduro.

9. De código abierto.

La comunidad de usuarios de OrientDB formada por 70 colaboradores y miles de usuarios en todo el mundo, contribuyen a mejorar la estabilidad del producto y a innovar para hacer de OrientDB un producto NoSQL más completo.

2.4 BENCHMARKS EXISTENTES ENTRE DISTINTOS SISTEMAS

Hoy en día existen numerosos benchmarks en los que se analiza el rendimiento de un sistema o se comparan las características y rendimiento de varios sistemas. En esta sección se mencionan algunos de ellos, como por ejemplo la comparación entre OrientDB y Neo4j [31] que realiza el equipo de Tecnologías Orient y algunos otros que se incluyen en [32].

2.4.1 ORIENTDB VS NEO4J[31]

OrientDB y Neo4j comparte muchas características pero la principal diferencia se encuentra en los motores. Neo4j es una BDOG pura mientras que OrientDB, posee un motor híbrido documento-grafo el cual añade ciertas características importantes al modelo basado en BDOG. A continuación se detallan las principales diferencias entre estos dos sistemas.

2.4.1.1 SISTEMA GESTOR DE BASES DE DATOS OPERACIONAL

Muchas soluciones NoSQL, se utilizan simplemente como "caché" para acelerar algunos casos de uso. La mayoría de productos NoSQL se centran en el rendimiento y la escalabilidad, mientras que la fiabilidad la dejan algo de lado. En este perfil de solución se encuadra Neo4j.

Sin embargo, OrientDB gracias a WAL (Write Ahead Logging), es capaz de restaurar el contenido de la base de datos después de cualquier accidente, como se veía en la sección *2.3.3.9.1 Principales ventajas* de OrientDB. Por lo que este sistema, además de tener un buen rendimiento y escalabilidad, también se acuerda de la fiabilidad marcando la diferencia con Neo4j.

2.4.1.2 ESCALABILIDAD

Neo4j soporta replicación pero en su versión de pago *Enterprise Edition*, no en la versión gratuita *Open Source Community Edition*. Además, la replicación sigue una arquitectura de tipo maestro/esclavo con un gran cuello de botella en operaciones de escritura. Solo un servidor puede ser el maestro, por lo que el rendimiento de escrituras en Neo4j se limita a la capacidad de ese servidor maestro. Esto significa que Neo4j no escala en las escrituras.

En la arquitectura que presenta OrientDB, todos los servidores son maestros, por lo que el rendimiento no está limitado a un único servidor. El rendimiento global es la suma de los rendimientos de todos los servidores. Esto también se conoce como escalabilidad linear.

2.4.1.3 LENGUAJE DE CONSULTA

Neo4j posee su propio lenguaje de consulta llamado Cypher, mientras que OrientDB utiliza un lenguaje basado en SQL complementado con algunas extensiones para manipular árboles y grafos.

[31] http://orientdb.com/orientdb-vs-neo4j/

2.4.1.4 DOMINIOS COMPLEJOS

Neo4j no dispone de esquema pero sí utiliza el concepto de "etiqueta" para agrupar nodos y relaciones del mismo tipo. Sin embargo, OrientDB permite crear esquemas alrededor de los grafos. Además ofrece servicios de herencia y polimorfismo, lo que permite crear subclases de nodos y relaciones.

2.4.1.5 CARACTERÍSTICAS GENERALES

A continuación se presenta una tabla en la que se indica si los sistemas Neo4j y OrientDB cumplen (en verde) o no (en rojo) con distintas características especificadas en la columna "Características".

Características	Neo4j	OrientDB
Base de datos orientada a grafos		
Conceptos de orientación a objetos		
Seguridad de roles y usuarios		
Bloqueo a nivel de registro		
Lenguaje Cypher		
Lenguaje SQL		
Transacciones ACID		
Recorridos por relaciones	O(1)	O(1)
Tipos de datos personalizados		
Replicación multi-maestro		
HTTP Rest/JSON nativo		
Funciones del lado del servidor		

Tabla 2: OrientDB vs Neo4j

2.4.2 OTROS BENCHMARKS

Como se ha comentado anteriormente, existen infinidad de estudios comparando distintos sistemas de BDOG entre ellos y con otros sistemas gestores de bases de datos relacionales. Seguidamente se incluyen referencias a algunos de ellos:

- ***Benchmarking traversal operations over graph databases***[32]: este estudio presenta los resultados de una prueba comparativa contra Neo4j, DEX (Sparksee), OrientDB, un repositorio RDF nativo y SGDB. Incluye operaciones como por ejemplo inserciones de elementos y recorridos locales y globales. Los resultados preliminares revelaron problemas de carga para grandes conjuntos de datos en las BDOG. Se obtuvo un bajo rendimiento en recorridos globales sobre grande redes. Sin embargo, el rendimiento se mantuvo estable en recorridos locales de 2 o 3 niveles de profundidad.

- ***A Comparison of a Graph Database and a Relational Database***[33]: este estudio compara MySQL con Neo4j para averiguar cuál de ellos es más adecuado para un sistema de procedencia de los datos. La comparación tiene en cuenta el tamaño de la base de datos y el espacio de disco requerido. Neo4j mostró un mejor rendimiento

[32] http://ups.savba.sk/~marek/papers/gdm12-ciglan.pdf
[33] http://www.cs.olemiss.edu/~ychen/publications/conference/vicknair_acmse10.pdf

que MySQL en la mayoría de las consultas. La BDOG fue mucho más rápida que la de tipo relacional (a veces fue hasta 10 veces más rápida) cuando se ejecutaron consultas sobre recorridos. No obstante, Neo4j requiere hasta dos veces el espacio que necesita MySQL, lo que significa un mayor espacio de disco.

3. ANÁLISIS DE LA SOLUCIÓN

Como se comentó en la introducción de este documento, el presente trabajo, además de consistir en una comparativa, en lo que se refiere a la extracción de información, entre Neo4j y RSHP, también incluye una parte de análisis y diseño de la forma en la que se almacenarán los datos en el grafo de Neo4j para realizar el experimento, otra parte de generación automática de consultas y una última parte de ejecución automática de consultas en Neo4j. En este capítulo se describen las distintas propuestas que se pensaron, desde el comienzo del proyecto hasta la solución final adoptada. Antes de nada, se incluye una especificación de requisitos correspondientes al software implementado.

3.1 ESPECIFICACIÓN DE REQUISITOS

Como se ha comentado anteriormente, este Trabajo de Fin de Grado consta de dos partes bien diferenciadas. Una de ellas correspondiente a ingeniería del software, en la que se implementa un software cuyo objetivo es la generación del grafo en Neo4j, a partir del cual se generan consultas y se ejecutan automáticamente en Neo4j. Este software se utilizará para la posterior comparativa entre Neo4j y RSHP. Por otro lado se incluye toda la parte de experimentación entre los dos sistemas.

En este apartado se incluye una especificación de requisitos para la parte correspondiente a la ingeniería del software. Para comprender bien los requisitos que se van a mostrar, cabe destacar que los datos utilizados para la posterior comparativa son un conjunto de requisitos. Asimismo, se considera un término a cada una de las palabras que conforman un requisito. A continuación se muestra una tabla de ejemplo tal y como se va a utilizar para cada uno de los requisitos:

Requisito software funcional	
Identificador	RF-XXX
Nombre	
Descripción	
Necesidad	
Prioridad	

Tabla 3: Tabla de ejemplo de especificación de requisito

Como se puede ver en la Tabla 3, por cada requisito se van a indicar los siguientes campos:

- **Identificador**: se trata de un código exclusivo y abreviado para cada requisito. Como se puede ver en la tabla anterior se construye mediante las letras RF (Requisito Funcional), seguidas de un guión y el número de requisito correspondiente formado por tres dígitos. Por lo que, el identificador del primer requisito será RF-001.

- **Nombre**: se corresponde con el título del requisito o la identificación extendida del mismo.

- **Descripción**: en este campo se detalla el requisito.

- **Necesidad**: en este campo se incluirá si el requisito debe cumplirse de forma obligatoria o si por el contrario no es imprescindible. A continuación se indican los dos valores que puede tomar:

  - ✓ **Esencial**: el requisito tiene que ser implementado de forma obligatoria.
  - ✓ **Recomendable**: preferiblemente se debe implementar el requisito, pero no es estrictamente obligatorio.

- **Prioridad**: define la importancia de un requisito, en base a la cual se implementará al comienzo o en fases más avanzadas. Los valores que puede tomar este campo son los siguientes:

  - ✓ **Alta**: el requisito debe implementarse en las fases más tempranas del desarrollo.
  - ✓ **Media**: el requisito debe implementarse cuando se hayan implementado todos los requisitos de prioridad alta.

Seguidamente, se incluyen los requisitos funcionales establecidos para el software implementado:

Requisito software funcional	
Identificador	RF-001
Nombre	Parámetros recibidos por el Generador del Grafo
Descripción	El Generador del Grafo recibirá el fichero con los requisitos a indexar y el fichero con los patrones.
Necesidad	Esencial
Prioridad	Alta

Tabla 4: Requisito RF-001 - Parámetros recibidos por el Generador del Grafo

Requisito software funcional	
Identificador	RF-002
Nombre	Creación del grafo en Neo4j
Descripción	El Generador de Grafo creará el grafo en Neo4j indexando las propiedades “id” y “texto” de cada uno de los requisitos en los nodos y relaciones del grafo.
Necesidad	Esencial
Prioridad	Alta

Tabla 5: Requisito RF-002 - Creación del grafo en Neo4j

Requisito software funcional	
Identificador	RF-003
Nombre	Envío del grafo al Generador de Consultas
Descripción	El Generador del Grafo enviará el grafo resultante al Generador de Consultas.
Necesidad	Esencial
Prioridad	Alta

Tabla 6: Requisito RF-003 - Envío del grafo al Generador de Consultas

Requisito software funcional	
Identificador	RF-004
Nombre	Consultas a generar por el Generador de Consultas
Descripción	El Generador de Consultas recibirá por parámetro el número de consultas que el usuario desee generar.
Necesidad	Esencial
Prioridad	Alta

Tabla 7: Requisito RF-004 - Consultas a generar por el Generador de Consultas

Requisito software funcional	
Identificador	RF-005
Nombre	Número mínimo y máximo de términos por consulta
Descripción	El Generador de Consultas recibirá dos parámetros en los que el usuario indicará el número mínimo y máximo de términos para cada consulta.
Necesidad	Esencial
Prioridad	Alta

Tabla 8: Requisito RF-005 - Número mínimo y máximo de términos por consulta

Requisito software funcional	
Identificador	RF-006
Nombre	Selección aleatoria del número de términos por consulta
Descripción	El Generador de Consultas formará las consultas seleccionando de forma aleatoria el número de términos de cada consulta (siempre dentro del rango especificado en los parámetros del requisito RF-005).
Necesidad	Recomendable
Prioridad	Media

Tabla 9: Requisito RF-006 - Selección aleatoria del número de términos por consulta

Requisito software funcional	
Identificador	RF-007
Nombre	Selección aleatoria de los términos incluidos en cada consulta
Descripción	El Generador de Consultas formará las consultas seleccionando los términos del vocabulario del grafo de forma aleatoria para cada consulta.
Necesidad	Recomendable
Prioridad	Media

Tabla 10: Requisito RF-007 - Selección aleatoria de los términos incluidos en cada consulta

Requisito software funcional	
Identificador	RF-008
Nombre	Términos no repetidos
Descripción	El Generador de Consultas no seleccionará dos veces un mismo término hasta que no hayan sido seleccionados todos los términos del vocabulario del grafo.
Necesidad	Recomendable
Prioridad	Media

Tabla 11: Requisito RF-008 - Términos no repetidos

Requisito software funcional	
Identificador	RF-009
Nombre	Consultas en Cypher y lenguaje natural
Descripción	El Generador de Consultas formará las consultas tanto en lenguaje Cypher (para Neo4j) como en lenguaje natural (para RSHP).
Necesidad	Esencial
Prioridad	Alta

Tabla 12: Requisito RF-009 - Consultas en Cypher y lenguaje natural

Requisito software funcional	
Identificador	RF-010
Nombre	División consultas para Neo4j
Descripción	El Generador de Consultas dividirá una consulta para Neo4j, que contenga términos almacenados en un nodo y términos almacenados en relaciones, en dos consultas. Una consulta será sobre nodos y la otra sobre relaciones.
Necesidad	Esencial
Prioridad	Alta

Tabla 13: Requisito RF-010 - División consultas para Neo4j

Requisito software funcional	
Identificador	RF-011
Nombre	Datos retornados por las consultas en Cypher
Descripción	Las consultas en lenguaje Cypher devolverán los identificadores de los requisitos indexados en el grafo.
Necesidad	Esencial
Prioridad	Alta

Tabla 14: Requisito RF-011 - Datos retornados por las consultas en Cypher

Requisito software funcional	
Identificador	RF-012
Nombre	Ficheros distintos consultas Neo4j y RSHP
Descripción	El Generador de Consultas escribirá las consultas para Neo4j y RSHP en ficheros distintos.
Necesidad	Esencial
Prioridad	Alta

Tabla 15: Requisito RF-012 - Ficheros distintos consultas Neo4j y RSHP

Requisito software funcional	
Identificador	RF-013
Nombre	Envío consultas al Receptor de Consultas
Descripción	El Generador de Consultas enviará el fichero con las consultas correspondiente a Neo4j al Receptor de Consultas.
Necesidad	Esencial
Prioridad	Alta

Tabla 16: Requisito RF-013 - Envío consultas al Receptor de Consultas

Requisito software funcional	
Identificador	RF-014
Nombre	Envío consultas al Motor de Base de Datos
Descripción	El Receptor de Consultas enviará las consultas, recibidas por el Generador de Consultas, al Motor de Base de Datos.
Necesidad	Esencial
Prioridad	Alta

Tabla 17: Requisito RF-014 - Envío consultas al Motor de Base de Datos

Requisito software funcional	
Identificador	RF-015
Nombre	Ejecución consultas Neo4j
Descripción	El Motor de Base de Datos (Neo4j) ejecutará las consultas recibidas por el Receptor de Consultas.
Necesidad	Esencial
Prioridad	Alta

Tabla 18: Requisito RF-015 - Ejecución consultas Neo4j

Requisito software funcional	
Identificador	RF-016
Nombre	Creación fichero para resultados en Neo4j
Descripción	El Motor de Base de Datos generará un fichero en el que se incluirán los resultados obtenidos por cada una de las consultas ejecutadas en Neo4j.
Necesidad	Recomendable
Prioridad	Media

Tabla 19: Requisito RF-016 - Creación fichero para resultados en Neo4j

3.2 CREACIÓN DEL GRAFO EN NEO4J

En primer lugar, cabe destacar que los datos utilizados para la comparativa son un conjunto de requisitos. Teniendo en cuenta esto, se va a proceder a detallar minuciosamente, tanto la propuesta inicial de la forma de almacenar esos requisitos en un grafo Neo4j, como las sucesivas propuestas hasta llegar a la solución final.

3.2.1 PROPUESTA INICIAL

Antes de indexar los requisitos en un grafo de Neo4j, había que pensar qué información se iba a almacenar y cómo hacerlo para realizar el experimento de forma satisfactoria. Desde el comienzo se decidió que los datos que se iban a almacenar iban a ser el texto de cada uno de los requisitos y su correspondiente identificador. No se almacenó más información ya que era irrelevante para la realización del experimento.

A partir de aquí, hubo que pensar cómo se iba a almacenar esta información en el grafo. En un primer momento, se decidió almacenar cada uno de los términos de cada requisito en un nodo (en una propiedad “id”), guardando en una propiedad “requisitos”, en cada nodo, los identificadores de los requisitos que contenían ese término. Cabe resaltar que en esta propuesta no se almacenaba ninguna propiedad en las relaciones entre nodos.

Además, en esta propuesta para la creación del grafo, los términos de los requisitos se insertaban en el grafo sin seguir ningún tipo de patrón, simplemente se recorrían los requisitos y se iban creando los nodos con los correspondientes términos. En el grafo resultante no había dos nodos con el mismo término, es decir, en el caso de que un término apareciese más de una vez, se actualizaba la propiedad “requisitos” del nodo en cuestión.

3.2.2 SEGUNDA PROPUESTA PLANTEADA

Después de esbozar la propuesta inicial y a partir de ella, se planteó la idea de almacenar los términos de los requisitos no sólo en los nodos, sino también en las relaciones. De esta forma se podrían recuperar tanto nodos como relaciones del grafo, definiendo un grafo más completo e interesante. Esto suponía un cambio importante a la hora de crear el grafo, ya que había que seguir recorriendo los requisitos de la misma forma que en la propuesta inicial, pero había que identificar qué términos iban a ser nodos y cuáles iban a ser relaciones.

En este capítulo se va a describir el proceso que se adoptó para crear el grafo en esta segunda propuesta. En los capítulos “Diseño” e “Implementación” se detallará más en profundidad dicho proceso. Básicamente, lo que se planteó fue incluir los términos que iban a ser relaciones en un fichero. De este modo, a la hora de crear el grafo se consultaría a este fichero si contiene el término o no, para así crear un nodo o una relación según corresponda. Como se ha comentado, en los capítulos “Diseño” e “Implementación” se completará la explicación de este proceso incluyendo un ejemplo final de grafo resultante y describiendo cómo se llevó a cabo desde el punto de vista programático, respectivamente.

En cuanto a la forma de almacenar la información, respecto de la propuesta inicial, se mantuvo que los nodos tendrían una propiedad “id” donde se almacenaría cada uno de los términos y una propiedad “requisitos” donde se almacenarían los identificadores de los requisitos que tuviesen el término correspondiente a ese nodo. Además, como en esta

segunda propuesta podría haber términos tanto en los nodos como en las relaciones, se añadieron las propiedades "id" y "requisitos" a todas las relaciones del grafo. Cabe destacar que un término sólo puede aparecer en un nodo o en relaciones, pero no puede aparecer en un nodo y relaciones al mismo tiempo.

3.2.3 DESCRIPCIÓN DE LA SOLUCIÓN FINAL

Finalmente, respecto a la segunda propuesta, se mantuvo la idea de almacenar términos tanto en nodos como en relaciones. Sin embargo, la forma en la que se identificaban las relaciones y se indexaban los datos, no era todo lo elegante que se quería, por lo que se decidió cambiar esa parte. En la solución final, el grafo se crea mediante **patrones**. A través de ellos, los requisitos son indexados en el grafo. Cabe destacar que cada patrón puede indexar un número importante de requisitos, por lo que no se necesitarán demasiados patrones para completar el grafo utilizado en el experimento.

Es importante resaltar dos aspectos significativos relacionados con los patrones. En primer lugar, los patrones poseen ciertos términos que son fijos y otros que pueden ser variables. Es decir, los términos fijos del patrón deben mapearse directamente con los términos del requisito para que éste se indexe de forma correcta. Mientras que los términos variables, pueden coincidir con el término del requisito o con cualquier otro, por lo que aceptan cualquier término en esa posición del patrón. A continuación se incluye un pequeño ejemplo:

Requisito 1: El coche debe arrancar
Requisito 2: El vehículo debe frenar
Requisito 3: El avión debe acelerar
Requisito 4: El sistema debe frenar

Patrón: El [sistema] debe [acción]

Figura 92: Términos fijos y variables en los patrones

Como se puede observar en la *Figura 92*, se tiene el patrón "El [sistema] debe [acción]" que generaría los cuatro requisitos que se ven más arriba. Los términos fijos serían "El" y "debe" que se mapean directamente entre el patrón y los requisitos (en las mismas posiciones), mientras que los términos variables serían "sistema" y "acción" (incluidos entre corchetes), en cuyas posiciones se admitiría cualquier término. En este caso, en la segunda posición del patrón, se admitirían los términos coche, vehículo, avión, sistema y cualquier otro término que se quisiese incluir. Si el segundo término del patrón no estuviese entre corchetes, es decir, que fuese fijo en lugar de variable, sólo se indexaría correctamente el *Requisito 4*, ya que sería el único que mapea de forma directa el mismo término tanto en el patrón como en el requisito.

El otro aspecto a remarcar tiene que ver con la identificación de los términos que serán nodos y los que serán relaciones. Esto es necesario indicarlo en los patrones de alguna forma, para que cuando se cree el grafo, sean ellos mismos los que tengan marcado qué términos serán nodos y cuáles relaciones. Esto se ha decidido marcar en los patrones indicando, antes de cada término, entre paréntesis una "r" (relación) o una "n" (nodo). Por lo que los patrones han sido marcados de la forma que se indica en la *Figura 93* para la creación del grafo en Neo4j.

Patrón: (n)El (n)[sistema] (r)debe (n)[acción]

Figura 93: Ejemplo de marcado de patrón

En el capítulo "Implementación" se detallará todo el proceso de indexación de los requisitos a través de los patrones, desde un punto de vista programático.

Por lo que se refiere a la forma de almacenar la información, se mantiene casi todo como se realizaba en la segunda propuesta. Es decir, tanto nodos como relaciones almacenan términos, teniendo las **propiedades "id" y "requisitos"** todos los nodos del grafo y **todas aquellas relaciones que tengan almacenado un término**. Es importante resaltar, que no todas la relaciones tienen estas dos propiedades ya que si un patrón (como ocurre en la *Figura 93*) tiene dos términos seguidos marcados como nodos, se crea una relación vacía entre ambos, ya que dicha relación no almacena ningún término. En el capítulo "Diseño", se incluirá un ejemplo de grafo en el que se podrá comprobar esta pequeña diferencia de la solución final con la segunda propuesta que se planteó.

3.3 GENERACIÓN AUTOMÁTICA DE CONSULTAS

El trabajo realizado también incluye un proceso de generación automática de consultas, para ser ejecutado después de forma programática mediante el API para Java de Neo4j. Cabe resaltar que las consultas que se generan son tanto las que se ejecutan en Neo4j como las que se ejecutan en KnowledgeMANAGER, las primeras en lenguaje Cypher y las segundas en lenguaje natural. Se ha incluido esta parte, para que el proceso de crear las consultas a ejecutar sea instantáneo y no se pierda tiempo en generar las consultas a mano.

Cabe resaltar que los términos que aparecen en las consultas se escogen de forma aleatoria del vocabulario del grafo en Neo4j. Además, no se admite un mismo término en una consulta ni tampoco se repite ningún término hasta que no se hayan seleccionado todos los términos del vocabulario en todas las consultas anteriores. Todas las decisiones de implementación se detallarán en el capítulo "Implementación". En este apartado se va a detallar cómo se generaban las consultas inicialmente (cuando sólo se almacenaban los términos en los nodos) y cómo se generan las consultas en la solución final adoptada (con términos en los nodos y en las relaciones).

3.3.1 PROPUESTA INICIAL

En un primer momento, cuando el grafo sólo almacenaba los términos en nodos, se generaban tantas consultas como el usuario solicitase. Simplemente se obtenía el vocabulario del grafo (todos los términos) y se iban seleccionando de forma aleatoria tantos términos por consulta como el usuario indicara. De esta forma se iba conformando cada consulta hasta obtener todas y cada una de ellas.

3.3.2 CAMBIOS EN LA PROPUESTA INICIAL

Con el cambio que se realizó en el grafo, de pasar de almacenar los términos sólo en los nodos, a almacenarlos tanto en nodos como en relaciones, el proceso de generación de consultas se vio afectado. Los dos cambios principales que se produjeron son los siguientes:

- A la hora de recopilar el vocabulario del grafo, hubo que modificar la consulta con la que se recuperaba, para recuperar también los términos de las relaciones. Se pasó de recuperar el grafo con una consulta a recuperarlo con dos consultas, una para los términos de los nodos y otra para los de las relaciones.

- Se decidió que para aquellas consultas que contuviesen tanto términos almacenados en nodos como en relaciones, se generasen dos consultas, una para los términos almacenados en nodos y otra para los almacenados en relaciones computando los resultados de ambas como si fuese una sola consulta. Se realizó esta división ya que se intentó crear una sola consulta, en la que se pretendían recuperar los requisitos de un término almacenado en un nodo y de un término almacenado en una relación, obteniendo unos resultados erróneos, combinados de una forma extraña y, por tanto, sin validez para realizar el experimento. Sin embargo, mediante dos consultas separadas, una para nodos y otra para relaciones, los resultados esperados se obtenían de forma correcta.

3.3.3 DESCRIPCIÓN DE LA SOLUCIÓN FINAL

Por lo que la solución final de la generación automática de consultas consiste en lo siguiente:

1. Se recopila el vocabulario del grafo mediante dos consultas, una para los términos almacenados en nodos y otra para los almacenados en relaciones. A continuación se muestran cada una de estas consultas:

```
MATCH (n)
RETURN n.id;
```

Figura 94: Consulta para recuperar vocabulario en nodos

```
Match () - [r] -> ()
WHERE NOT r.id='null'
RETURN distinct r.id;
```

Figura 95: Consulta para recuperar vocabulario en relaciones

Como se puede comprobar en la *Figura 94*, como no existen dos nodos con el mismo término, simplemente se recuperan los términos, devolviendo la propiedad "id" de cada nodo (donde se almacenan los términos). Sin embargo, como se observa en la *Figura 95*, en la consulta para recuperar el vocabulario de las relaciones, se incluye una cláusula *WHERE* en la que se eliminan aquellas relaciones vacías (no tienen término almacenado) y en la cláusula *RETURN* se incluye la cláusula *distinct* para que cada término, sólo se devuelva una vez y así, obtener el vocabulario completo, sin repeticiones ni campos nulos.

2. Después, según el número de consultas que el usuario haya indicado, se van conformando. Por cada consulta se van seleccionando de forma aleatoria los términos del vocabulario recuperado. Se ha decidido que cada consulta tendrá entre 1 y 3 términos, una media aproximada del número total de consultas que se realizan cada día en Google [33]. A medida que se seleccionan los términos, se van construyendo las consultas, tanto para Neo4j como para KnowledgeMANAGER.

3. En el caso que para una consulta se hayan seleccionado tanto términos almacenados en nodos como en relaciones, la consulta para Neo4j se divide en dos, como se ha explicado en el apartado anterior, para recuperar los requisitos de forma correcta. A continuación se incluye un ejemplo:

```
-Términos seleccionados para la consulta:
--Almacenados en nodos: sistema, frenar
--Almacenados en relaciones: debe

-Consulta única para términos en nodos y relaciones (se obtienen resultados erróneos)
MATCH (n), ()-[r]->()
WHERE n.id='sistema' OR n.id='frenar' OR r.id='debe'
RETURN distinct n.requisitos, r.requisitos;

-Consulta para los términos en nodos
MATCH (n)
WHERE n.id='sistema' OR n.id='frenar'
RETURN distinct n.requisitos;

-Consulta para los términos en relaciones
MATCH ()-[r]->()
WHERE r.id='debe'
RETURN distinct r.requisitos;
```

Figura 96: División de aquellas consultas que tengan términos en nodos y relaciones (sólo para Neo4j)

Como se ha comentado anteriormente, la primera consulta que aparece en la *Figura 96*, arroja unos resultados que no son los esperados, ya que se pretenden recuperar propiedades de nodos y relaciones a la vez, filtrando por términos almacenados en nodos y relaciones. Sin embargo, en estos casos se divide la consulta en dos, separando la recuperación de los nodos de la de las relaciones. En el ejemplo anterior se pueden observar las dos consultas resultantes de dividir la primera consulta. En el caso de que para una consulta, sólo se seleccionasen términos almacenados en nodos o en relaciones, no habría que dividir la consulta, ya que no se mezclan términos guardados en nodos y en relaciones.

Cabe destacar que para ambos tipos de consultas, **se emplea el operador OR** cuando la consulta incluye más de un término. Se ha decidido utilizar este operador ya que es el que recupera los datos de forma correcta. No se utiliza el operador AND ya que no se obtendría ningún resultado para las consultas que lo utilizasen. Esto es debido a que un nodo sólo puede tener una propiedad "id", de modo que si se construye una consulta del tipo: *MATCH (n) WHERE n.id='sistema' AND n.id='frenar' RETURN distinct n.requisitos;* se está preguntando por un nodo que tenga una propiedad "id" sistema y otra frenar. Esto no devolvería ningún resultado, por lo que la consulta no es válida. Por este motivo, el operador utilizado en las consultas para Neo4j, con más de un término, es OR.

3.4 EJECUCIÓN AUTOMÁTICA DE CONSULTAS EN NEO4J

Como se ha explicado en el apartado anterior, se generan consultas tanto para Neo4j como para KnowledgeMANAGER. Las consultas para este último se ejecutan a mano, pero las correspondientes para Neo4j, se han decidido ejecutar automáticamente, de forma programática. De esta forma se crea un entorno automático de ejecución de consultas en Neo4j con el que se ahorra bastante tiempo, en comparación si se ejecutaran a mano. Se pueden ejecutar tantas consultas como se quieran de una sola vez, separando los resultados de cada una de ellas de forma clara.

En el capítulo “Implementación” se describen las decisiones tomadas en la programación de este proceso.

4. DISEÑO DEL GRAFO EN NEO4J

En este capítulo se van a detallar los distintos diseños de grafo en Neo4j que se han ido utilizando a lo largo del proyecto hasta llegar a la solución final. En primer lugar, se va a describir el diseño de la solución final para, posteriormente, comentar las diferentes alternativas que se han barajado antes de llegar a esa solución final. Para cada diseño se indicarán sus ventajas e inconvenientes y se explicarán las razones que llevaron a pasar de un diseño a otro hasta el diseño final.

4.1 DISEÑO DE LA SOLUCIÓN FINAL

Primeramente, se va a detallar el diseño final del grafo en Neo4j, prestando especial atención a la forma en la que se almacenan los requisitos en el grafo. Teniendo en cuenta el patrón que se ha visto en la *Figura 93* y los requisitos de la *Figura 92*, el grafo que se generaría mediante ese patrón es el siguiente:

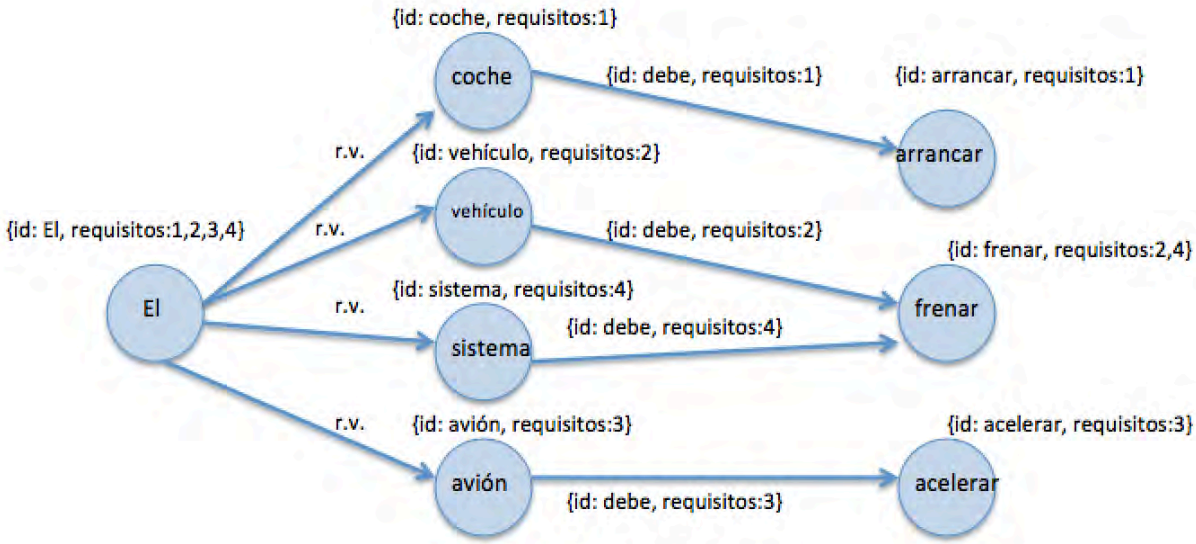


Figura 97: Ejemplo del diseño de la solución final (I)

En la *Figura 97* se puede observar un ejemplo de grafo generado mediante patrones, la forma en la que se crea el grafo en la solución final. En este caso, se comprueba como se han indexado correctamente los cuatro requisitos de la *Figura 92* mediante el patrón de la *Figura 93*.

En primer lugar, se puede comprobar como los términos se almacenan tanto en nodos como en relaciones. Asimismo, se observa como cada uno de los nodos posee las propiedades "id" y "requisitos", almacenando el término correspondiente en la propiedad "id" y los requisitos en los que aparece cada término en la propiedad "requisitos" de cada nodo.

En cuanto a las relaciones, se puede observar que hay dos tipos. Las relaciones que almacenan términos y las que no, marcadas con r.v. (relación vacía). Se comprueba que las relaciones que sí almacenan términos, también tienen las mismas propiedades que los nodos. Merece mención especial la particularidad de que no existen dos nodos con el mismo término (se acumulan los requisitos que tienen ese término en la propiedad "requisitos"), pero sí que existen varias relaciones con el mismo término, almacenando cada una de ellas el requisito correspondiente. Se ha tomado esta decisión en las relaciones ya que no todas las que tienen

el término “debe” enlazan los mismos nodos siempre, por lo que se tendrán varias relaciones que almacenen el mismo término, con el respectivo requisito almacenado en cada una de ellas. Incluso si un término almacenado en una relación enlaza los dos mismos nodos varias veces, se crean tantas relaciones como veces enlace esos nodos. A la hora de recuperar un término, almacenado en distintas relaciones como es el caso, simplemente se realizaría la consulta pertinente, filtrando en la cláusula WHERE por el término en cuestión y devolviendo la propiedad “requisitos”. A continuación se indica la consulta en este ejemplo:

```
Match () - [r] -> ()
WHERE r.id='debe'
RETURN distinct r.requisitos;
```

Figura 98: Consulta recuperando los requisitos de un término almacenado en relaciones

En este caso, al ejecutar la consulta de la *Figura 98*, se obtendrían los cuatro requisitos, más concretamente se recuperarían sus identificadores: *1,2,3,4*.

En el ejemplo anterior, se ha comprobado cómo es el diseño final cuando el patrón y los requisitos se mapean directamente, es decir, cuando tienen ambos el mismo número de términos. Seguidamente se incluye un ejemplo en el que el patrón es más corto que los requisitos, para comprobar qué sucede con el exceso de términos de los requisitos. El ejemplo es el siguiente:

Requisito 1: El coche debe poder arrancar
Requisito 2: El coche debe poder arrancar con huella dactilar
Requisito 3: El vehículo podrá frenar automáticamente
Requisito 4: El avión debe poder acelerar de 0 a 100 en 2 segundos
Requisito 5: El sistema debe poder frenar

Patrón: (n)El (n)[sistema] (r)debe (r)poder (n)[acción]

Figura 99: Ejemplo de mapeo no directo entre patrón y requisito

En la *Figura 99* se observa el nuevo ejemplo y se comprueba como se ha añadido una nueva relación fija “poder”, antes del nodo final del patrón, se ha añadido un nuevo requisito y se han modificado ligeramente los demás. En la siguiente figura se va a mostrar el grafo resultante de indexar esos requisitos con el nuevo patrón. Posteriormente, se detallarán las decisiones de diseño que no hayan sido explicadas y se indicará qué requisitos se indexan correctamente y cuáles no, justificando cada una de las explicaciones.

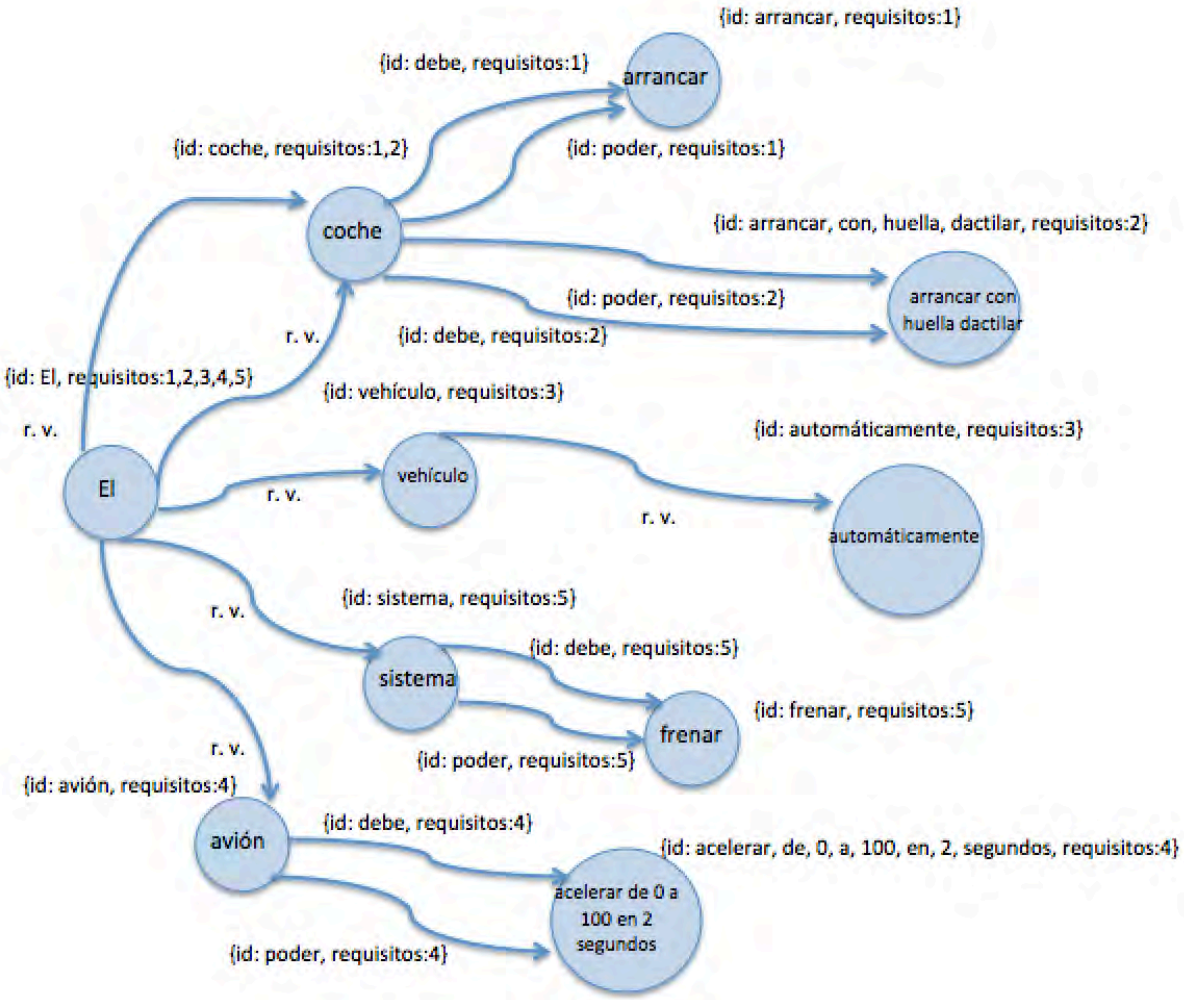


Figura 100: Ejemplo del diseño de la solución final (II)

En la *Figura 100* se observa el grafo resultante de indexar los requisitos de la *Figura 99* a través del patrón que aparece en esa misma figura. Con este ejemplo se completa la explicación del diseño de la solución final iniciada con el ejemplo anterior. Principalmente, cabe destacar tres decisiones importantes en el diseño, que en el anterior ejemplo no se vieron y se detallan a continuación:

- Cuando se da la circunstancia de que el requisito tiene más términos que el patrón, como ocurre en los requisitos 2 y 4 del ejemplo, hasta el penúltimo término del patrón se mapean todos los términos de forma directa con el requisito. A partir del último término del patrón, **se acumula el exceso de términos en el último nodo**. Como se puede comprobar en la *Figura 100*, existen dos nodos que en la propiedad "id" almacenan varios términos debido a este motivo. Aunque en estos nodos se almacenen varios términos, a la hora de extraer el vocabulario para generar las consultas, ese conjunto de términos que posee un nodo computa como un único término.

- Es importante explicar por qué el requisito 3 no se ha indexado de forma correcta. Como se puede observar en el grafo anterior, todos los requisitos se han indexado de forma correcta excepto el tercero. Como se veía anteriormente, el requisito 3 tiene los mismos términos que el patrón, pero su tercer y cuarto término ("podrá frenar") no se corresponden con los términos fijos 3 y 4 del patrón ("debe poder"). Si esos términos

del patrón fuesen variables (entre corchetes), el requisito 3 se habría indexado correctamente, pero al ser fijos y no corresponder con los términos del requisito 3, esas relaciones de ese requisito no se han añadido al grafo. Por consiguiente, el último término del requisito sí se inserta de forma correcta incluyendo una relación vacía entre el anterior nodo y él, ya que no se han podido añadir las relaciones por el motivo explicado. Por lo que, del requisito 3 sólo se han insertado de forma correcta los términos "El", "vehículo" y "automáticamente". Esto obliga a prestar especial cuidado en la elección de que patrón genera qué requisitos ya que en este caso, si se consultara por el término "frenar" sólo se devolvería el requisito 5, teniendo que devolverse también el requisito 3 ya que posee ese mismo término.

- Otro aspecto fundamental en el diseño de la solución final es que un término sólo se puede almacenar en un nodo o en relaciones. No se podrá encontrar un término almacenado tanto en un nodo como en relaciones al mismo tiempo. Esto simplifica el proceso de recuperación del vocabulario en la generación de consultas ya que, un término estará en un nodo o en relaciones, por lo que sólo se generará una consulta para ese término, de nodos o de relaciones.

Además, como el patrón incluye varias relaciones entre dos nodos, se comprueba que se insertan en el grafo tantas relaciones entre nodos como relaciones se indican en el patrón (siempre que sean los mismos términos tanto en el requisito como en el patrón, si son fijos). Por último, comentar que si se da la situación de que un requisito tiene menos términos que el patrón que lo indexa, simplemente el requisito se indexa hasta llegar a su último término, aunque esta posibilidad es muy poco frecuente.

La principal ventaja que ofrece este modelo es la facilidad a la hora de indexar el corpus en el grafo. Identificando unos pocos patrones (en los que ya se indica qué términos serán nodos y cuáles relaciones), se puede llegar a indexar un volumen de datos considerable. Si se escogen unos buenos patrones, que indexen los datos sin acumular términos en los nodos (es decir, que la longitud del patrón sea mayor o igual a la de los requisitos que genera), este diseño recupera los datos esperados de forma excelente.

En cuanto a las desventajas, la más reseñable es la acumulación de términos en un único nodo cuando el requisito es de mayor longitud que el patrón que lo genera. Como se ha visto anteriormente, esto provoca que para un determinado término, quepa la posibilidad de que no se devuelvan todos los identificadores correspondientes a los requisitos esperados. Asimismo, en este diseño existe una desproporción entre el número de nodos y relaciones, siendo el de estas últimas mucho mayor.

4.2 ALTERNATIVAS DE DISEÑO

En este apartado se van a detallar otros diseños válidos pero descartados a lo largo del desarrollo del proyecto, como se comentó anteriormente en el capítulo de "Análisis" indicando las razones por las que se cambió de diseño del grafo. En primer lugar se explicará el diseño inicial con el que se comenzó, para continuar después con el segundo que se planteó el cual precedió al diseño adoptado en la solución final.

4.2.1 DISEÑO INICIAL

En este apartado se van a utilizar los requisitos de la *Figura 99* para comprobar cuál era el diseño del grafo que se realizó en primera instancia. Cabe recordar que en este primer diseño, los términos de los requisitos sólo se almacenaban en nodos y no se utilizaban patrones para generar el grafo. Simplemente se iban recorriendo los requisitos, creando los distintos nodos del grafo. El grafo resultante de indexar los requisitos mencionados anteriormente es el siguiente:

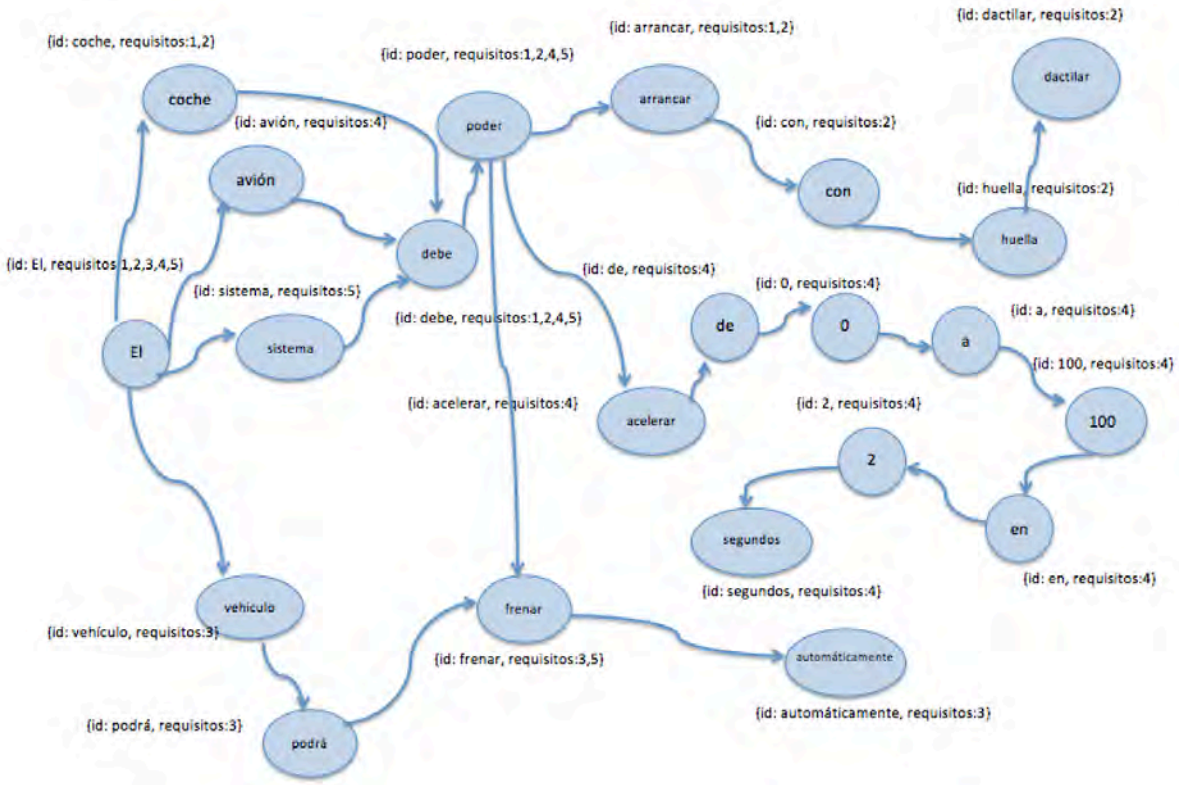


Figura 101: Diseño inicial

A partir de la *Figura 101* se comprueba como los términos sólo son almacenados en los nodos del grafo. En este diseño, cada nodo también tiene las propiedades "id" y "requisitos" y las relaciones en este caso, son todas vacías ya que no almacenan ningún término. Además, un mismo término sólo se almacena en un único nodo, acumulando los requisitos en los que aparece en su propiedad "requisitos". Se trata de un diseño muy simple, en el que se generan gran cantidad de nodos.

La principal ventaja de este primer diseño es precisamente su simplicidad. Gracias a ella, es el diseño que mejor recupera los requisitos relevantes para las consultas en Neo4j. Sus principales desventajas son el gran número de nodos que se generan y todas sus relaciones son vacías, desaprovechando un espacio importante donde también se pueden almacenar términos.

Esta opción es totalmente válida, pero en este trabajo se quiso añadir algo de complejidad permitiendo almacenar los términos tanto en nodos como en relaciones. Fue por esta razón por la que se pasó de este primer diseño al que se verá en el siguiente apartado.

4.2.2 SEGUNDO DISEÑO

Después de analizar el diseño inicial y confirmar que se quería añadir algo de complejidad a la solución final, se llegó a un segundo diseño en el que tanto nodos como relaciones, almacenaban términos. En este diseño tampoco se generaba el grafo mediante patrones, simplemente se recorrían los requisitos creando los nodos y relaciones del grafo. A continuación se va a mostrar el grafo resultante de indexar los mismos requisitos que en el diseño inicial y en el segundo ejemplo del diseño final de la solución para comprobar las diferencias existentes entre ellos.

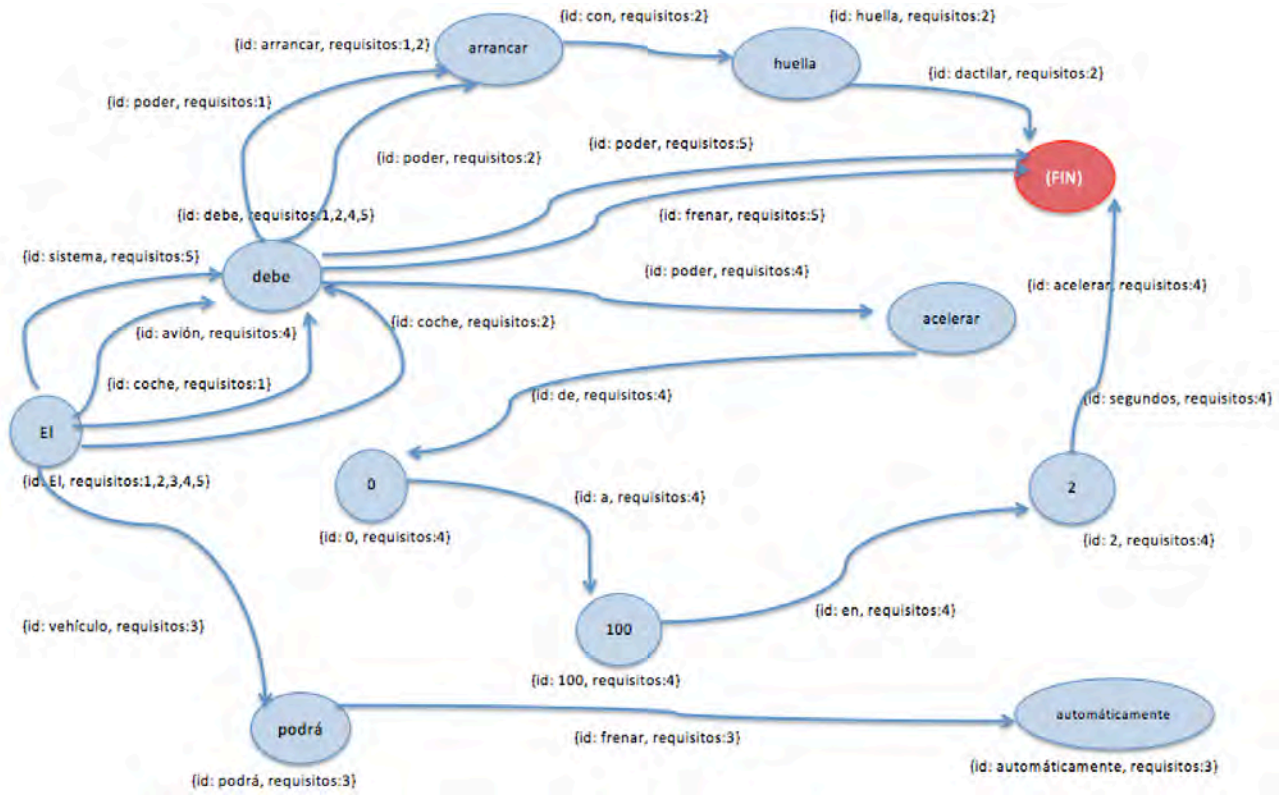


Figura 102: Segundo diseño planteado

En la *Figura 102* se puede observar el diseño del grafo que se pensó después del inicial. Como se ha comentado anteriormente, en este diseño, al igual que el inicial, tampoco se utilizan patrones para generar el grafo. Pero en este caso, como puede haber términos en los nodos y en las relaciones, hay que identificar qué términos serán las relaciones del grafo. Para ello, lo que se hace es, antes de empezar a generar el grafo, se recorren todos los requisitos identificando las relaciones, de modo que en este ejemplo las relaciones son los términos marcados en azul en la siguiente figura:

Requisito 1: El coche debe poder arrancar
Requisito 2: El coche debe poder arrancar con huella dactilar
Requisito 3: El vehículo podrá frenar automáticamente
Requisito 4: El avión debe poder acelerar de 0 a 100 en 2 segundos
Requisito 5: El sistema debe poder frenar

Figura 103: Identificando relaciones en el segundo diseño

A partir de la *Figura 103*, se comprueba como se identifican las relaciones en un conjunto de requisitos para crear posteriormente el grafo. En cada requisito, los nodos y relaciones se intercalan hasta llegar al último término de cada requisito. Este proceso de identificación de relaciones se realiza antes de comenzar a generar el grafo, de modo que cuando se empieza a crear, ya se conocen los términos que se almacenarán en las relaciones del grafo.

Puede darse la situación (como sucede en este caso) de que después de finalizar el proceso de identificación, un mismo término haya sido marcado como relación y a su vez como nodo. Esto sucede en el ejemplo con el término "frenar". Cabe resaltar que, cuando un término es marcado como relación en un requisito, siempre actuará como relación en el resto de requisitos en los que aparezca, ya que un mismo término no se almacenará nunca en un nodo y relaciones a la vez. Por lo que en el requisito 5 se tendrán tres relaciones: "sistema", "poder" y "frenar".

A raíz de esto, se genera un nuevo problema ya que un requisito no puede acabar en relación, debe terminar siempre apuntando a un nodo. Esto sucede tanto en el requisito 5, que se veía anteriormente, como en los requisitos con número de términos par (en el ejemplo los requisitos 2 y 4). Para solucionar este problema, se tomó la decisión de incluir, en este diseño, un nodo auxiliar (en la *Figura 102*, el nodo de color rojo), el cual no tiene ninguna propiedad y cuya función se limita a permitir que esos requisitos, que acaben en relación, apunten a él para que el grafo se genere correctamente. En el caso del requisito 5, al haber dos términos seguidos almacenados en relaciones, se generan dos relaciones entre el nodo "debe" y el nodo auxiliar "(FIN)".

Como sucedía en el diseño inicial, cada nodo posee las propiedades "id" y "requisitos". En este caso, al almacenar también términos en las relaciones, estas adquieren dichas propiedades, como sucede en el diseño final. Un aspecto importante de este diseño es la utilización que se hace de las relaciones. Como se puede comprobar en la *Figura 102*, todas las relaciones del grafo almacenan términos. En este diseño no se emplean relaciones vacías, como sucede en el diseño final, lo que permite que el grafo esté completo (todas las relaciones y nodos con términos, excepto el nodo auxiliar).

La principal ventaja de este diseño es su capacidad para aprovechar todo el potencial de almacenamiento del grafo. Aprovecha todas y cada una de las relaciones y nodos del grafo (excepto el nodo auxiliar) para almacenar los distintos términos de los requisitos. Otra ventaja importante es la alta efectividad que ofrece a la hora de recuperar los requisitos relevantes a una consulta en Neo4j.

Por lo que se refiere a las desventajas, cabe destacar la excesiva generación de relaciones con respecto al número de nodos presentes en el grafo. Asimismo, tener que incluir nodos auxiliares en los grafos, para que los requisitos que acaben con una relación se indexen de forma correcta, no es característico de un diseño fino y elegante.

Al igual que el diseño inicial, este segundo diseño que se planteó es totalmente válido para este trabajo, pero se decidió modificarlo, por el diseño finalmente adoptado, para obtener un diseño mucho más elegante, el cual también recupera los datos esperados de forma excelente. Se decidió realizar la creación e indexación del grafo mediante patrones.

Esto, a su vez, solucionaba el problema que se veía, unas líneas más arriba, de los requisitos pares que acababan en relación. Los patrones siempre empezarán y acabarán en un nodo, por lo que, con el nuevo diseño que se planteaba y que a posteriori ha resultado ser el definitivo, ese problema se solucionaba. En definitiva, la forma de almacenar los términos en los nodos y relaciones no varió del segundo diseño al definitivo, pero sí la forma en la que se generaba el grafo y la forma de identificar qué términos eran nodos y cuáles relaciones.

4.3 DIAGRAMAS DE COMPONENTES Y SECUENCIA DEL PROCESO DE AUTOMATIZACIÓN

Este apartado se divide en dos secciones. En la primera de ellas, se identifican y representan los componentes del proceso de automatización (creación del grafo, generación y ejecución de consultas), mientras que en la segunda sección se incluye un diagrama de secuencia en el que se puede comprobar cuál es el flujo de ejecución.

4.3.1 ARQUITECTURA DE COMPONENTES DEL PROCESO

Durante las fases de análisis y diseño, se han identificado una serie de componentes en los que se podría resumir el proceso de automatización. Estos componentes son los que se muestran en el siguiente diagrama de componentes:

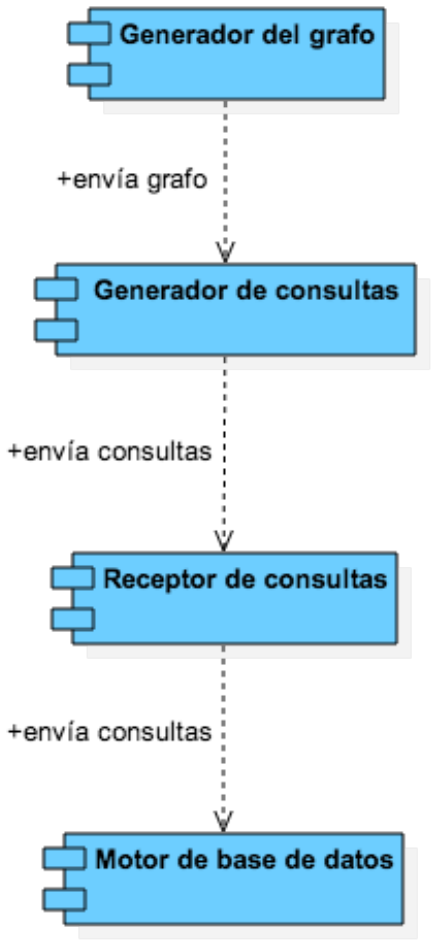


Figura 104: Diagrama de componentes

En la *Figura 104* se observa el diagrama de componentes, el cual ha sido generado mediante la herramienta StarUML[34] en su versión 2.5.0 para Mac OS X. Como se puede observar, se han identificado cuatro componentes los cuales son:

- **Generador del grafo:** la función de este componente es la creación del grafo y la correspondiente indexación de los datos en sus nodos y relaciones. Además, una vez generado el grafo, este componente lo envía al componente *Generador de consultas*.

- **Generador de consultas:** este componente recibe el grafo de del componente *Generador del grafo*. Es el encargado de extraer el vocabulario y de generar la consultas, las cuales se las pasa al componente *Receptor de consultas*.

- **Receptor de consultas:** este componente es el encargado de recibir las consultas del componente *Generador de consultas* y reenviárselas al componente *Motor de base de datos*.

- **Motor de base de datos:** Este componente se corresponde, en este caso, con Neo4j, quien recibe las consultas del componente Receptor de consultas y se encarga de ejecutarlas.

4.3.2 SECUENCIACIÓN DEL PROCESO

En esta sección se va a detallar cuál es el flujo de ejecución durante el proceso. Para ello, se incluye un diagrama de secuencia, realizado también mediante la herramienta StarUML:

[34] http://staruml.io/

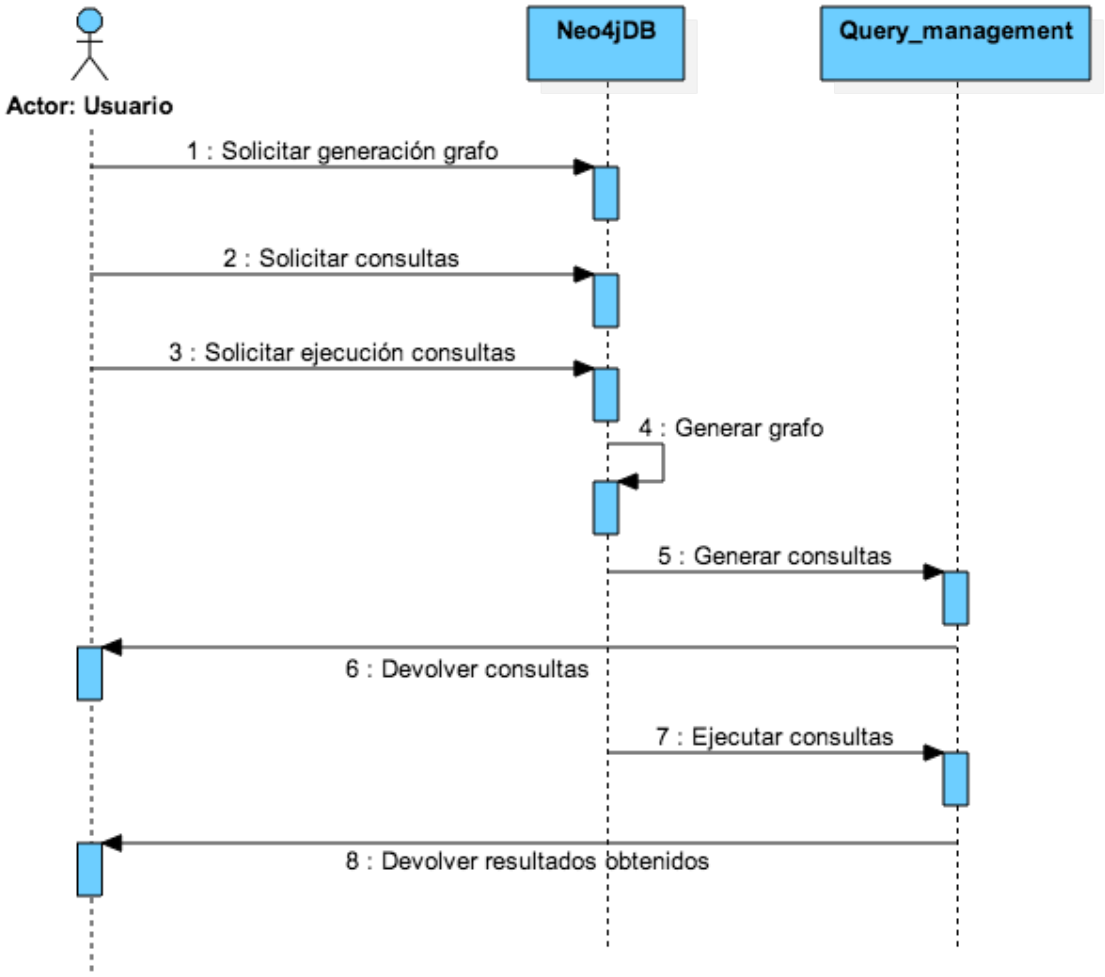


Figura 105: Diagrama de secuencia - Flujo de ejecución

A partir de la *Figura 105*, se comprueba como, en primer lugar, el usuario solicita que se genere el grafo, se generen el número de consultas que desee y se ejecuten (pasos 1, 2 y 3). Seguidamente, se crea el grafo (paso número 4), se generan las consultas y se devuelven al usuario (pasos 5 y 6). Finalmente, se ejecutan automáticamente las consultas generadas, devolviendo al usuario los resultados obtenidos (pasos 7 y 8). En caso de que el grafo ya esté formado, los pasos 1 y 4 no serían necesarios.

5. IMPLEMENTACIÓN

En este capítulo se va a detallar cómo ha ido evolucionando la implementación según los distintos diseños que se han ido adoptando a lo largo del proyecto. El **lenguaje de programación** utilizado para el desarrollo ha sido **Java** y se ha utilizado el **API para Java que ofrece Neo4j**[35], para poder manejar los grafos de forma programática. Cabe resaltar que para la implementación, tanto de la indexación en el grafo de Neo4j como de la generación y ejecución automática de consultas, se ha utilizado la herramienta **Eclipse** [36], más concretamente la versión eclipse Luna (4.4.1).

En primer lugar se va a detallar la implementación correspondiente a la indexación en el grafo de Neo4j, indicando los cambios sufridos por la implementación inicial hasta llegar a la implementación de la solución final. Seguidamente, se explicará el proceso que se sigue para generar y ejecutar las consultas automáticamente en Neo4j.

Cabe destacar que los métodos que se han implementado se han distribuido en dos clases, una de ellas incluye los métodos para la generación y ejecución de consultas (*Query_management.java*) y la otra incluye los correspondientes métodos para realizar la creación e indexación de los datos en el grafo y el método *main* para realizar la ejecución (*Neo4jDB.java*). Asimismo, es importante destacar que los requisitos se encuentran en una hoja Excel que incluye dos columnas, una para los identificadores de los requisitos y otra para el texto de los requisitos.

5.1 CREACIÓN DEL GRAFO

En este apartado se van a describir los métodos que se han utilizado para crear el grafo, tanto para el diseño de la solución final como para el resto de diseños analizados.

5.1.1 IMPLEMENTACIÓN INICIAL

La implementación que se realizó al comienzo, indexaba los términos de los requisitos sólo en los nodos del grafo. Cabe destacar que todas las relaciones creadas en el grafo, poseen un tipo de relación denominado *IS_REQUIREMENT*. En esta primera aproximación, se utilizó un único método que no recibía ningún parámetro, el cual se procede a detallar a continuación:

- **void createDb():** en este diseño inicial en el que los términos sólo se almacenan en nodos, este método fundamentalmente, realiza lo siguiente:

    1. En primer lugar, se recuperan todos los datos del Excel a través del API, para Java para documentos Microsoft, **Apache POI**[37]. Todos esos datos recuperados del Excel, se guardan en un ArrayList, almacenando en cada posición el identificador y el texto de cada requisito.

    2. En segundo lugar, se recorre el ArrayList y por cada requisito, se separa el identificador del texto, almacenando el primero en una variable String *idRequisito* y los términos del requisito en un array de String

[35] http://neo4j.com/docs/stable/javadocs/index.html
[36] https://eclipse.org/
[37] https://poi.apache.org/

arrayTextoRequisito en el que cada posición es un término. Este array será el que se recorra después para ir creando los distintos nodos del grafo.

3. Por último, se recorre de dos en dos el array que contiene los términos del requisito mediante un bucle anidado, para así ir creando los nodos y relaciones y completar todo el grafo con todos los términos de todos los requisitos. En la *Figura 106* se incluye un ejemplo de secuencia de cómo se van creando los nodos y relaciones de un requisito.

Figura 106: Ejemplo de funcionamiento del algoritmo de creación del diseño inicial

Como se puede apreciar en la *Figura 106*, los nodos y relaciones que aparecen en verde son los que se crean en esa iteración, mientras que los de color gris ya han sido creados en iteraciones anteriores. De esta forma se recorren todos los requisitos dando como resultado el grafo final.

5.1.2 CAMBIOS EN LA IMPLEMENTACIÓN INICIAL

Cuando se pasó del diseño inicial al segundo que se planteó, hubo que modificar la implementación para que se adaptara y generara el grafo teniendo en cuenta el nuevo diseño. Cabe recordar que este diseño almacenaba los términos tanto en nodos como en relaciones. A continuación se indican los nuevos métodos que se tuvieron que implementar y cómo se remodeló el método createDb() para este nuevo diseño:

- **void generateRelationships(String dataPath):** recibe un parámetro en el que se debe indicar la ruta de la ubicación del Excel con los requisitos. El propósito de este método es identificar qué términos serán las relaciones del grafo. El resultado de la ejecución de este método será un fichero que contenga dichos términos.

A diferencia de la primera implementación, en este caso se crean tres tipos de relaciones, OPCIONAL (se corresponde con el valor 0), RECOMENDABLE (valor 1) y ESENCIAL (valor 2). Esto no es relevante para lo que se pretende, que es recuperar los términos de grafo, pero permite que el grafo final sea más atractivo. Seguidamente, se recuperan los requisitos del Excel y se recorren, seleccionando primero de forma aleatoria, uno de los tres tipos de relación (0, 1 ó 2) para las relaciones de cada requisito. Cada requisito se recorre empezando por el segundo término de cada uno de ellos y saltando de dos en dos (ver *Figura 103*), de modo que los términos a los que se va accediendo serán las futuras relaciones del grafo.

Esos términos se van almacenando en un Hashtable en el que la clave son los propios términos y el valor el tipo de relación que les haya tocado a cada uno. En el caso de que un mismo término aparezca en varios requisitos y se le hayan asignado tipos de relación distintos, se escogerá el tipo de mayor relevancia siendo ESENCIAL > RECOMENDABLE > OPCIONAL. Una vez que se hayan recopilado todas las relaciones con sus tipos de relación, se escriben en un fichero denominado *relacionesGrafo.txt,* el cual se pasará como parámetro en el método

createDb y así, se podrá consultar si un término debe almacenarse en un nodo o en una relación. A continuación se incluye una imagen en la que se muestra como son escritos los términos y sus tipos de relación en el fichero generado:

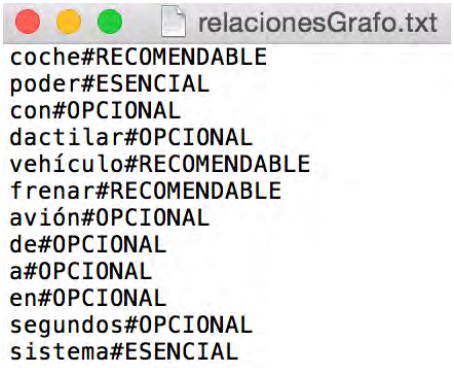


Figura 107: Formato de las relaciones y sus tipos de relación (diseño 2)

Como se puede observar en la *Figura 107*, se incluyen las relaciones identificadas en la *Figura 103*. Cada término se escribe en una línea del fichero, incluyendo cada uno el tipo de relación con el que se creará en el grafo. Se incluye una almohadilla (#) entre cada término y su correspondiente tipo de relación para que el procesamiento del fichero sea mucho más sencillo.

- **String searchTermInFile(String filePath, String term):** la función de este método consiste en lo siguiente: dado un fichero (recibido por parámetro), busca si un término (también recibido por parámetro) está dentro de ese fichero. Si se encuentra el término en el fichero, el método devuelve la línea en la que se ha encontrado y si no se encuentra, se devuelve una cadena de texto vacía. Este método será muy útil cuando se esté creando el grafo, para consultar si un término debe almacenarse en un nodo o en una relación. En este caso, se buscará un término en el fichero relacionesGrafo.txt y si se encuentra, como se devuelve la línea completa, se devolverá el término junto con el tipo de relación con el que se creará en el grafo.

- **void createDb(String dataPath, String relationshipsPath):** este método, en este segundo diseño, recibe dos parámetros; el primero para indicar la ruta del Excel que contiene los datos a indexar y el segundo que incluye la ruta del fichero que alberga los términos que serán relaciones en el grafo. Con respecto a este mismo método pero de la implementación inicial, se mantienen los dos primeros puntos que se veían en el apartado anterior. Además, se añade la siguiente funcionalidad:

    - ✓ Se recorre, en un bucle anidado, el array que contiene los términos del requisito que corresponda (según la iteración) de cuatro en cuatro términos, creando los nodos y relaciones que compondrán el grafo. Cabe destacar que en este caso, cada vez que se va a almacenar un nuevo término en el grafo, se consulta el fichero de relaciones, recibido por parámetro, para comprobar si se almacenará en un nodo o en una relación. Para realizar esa consulta, se utiliza el método *searchTermInFile(String filePath, String term)* explicado anteriormente. Si se encuentra el término, se crea en el grafo como relación, incluyendo su identificador de requisito en la propiedad "requisitos" y el tipo de relación que tenga asignado como se aprecia en la siguiente figura:

```
if(arrayRelacion[1].equals("ESENCIAL")){
    r2 = nodeInicioReq.createRelationshipTo(firstNode, RelTypes.ESENCIAL);
}
else if(arrayRelacion[1].equals("RECOMENDABLE")){
    r2 = nodeInicioReq.createRelationshipTo(firstNode, RelTypes.RECOMENDABLE);
}
else{
    r2 = nodeInicioReq.createRelationshipTo(firstNode, RelTypes.OPCIONAL);
}

r2.setProperty("id", arrayTextoRequisito[relacionesAcumuladas.get(z)]);
r2.setProperty("requisitos", idRequisito);
```

Figura 108: Creando una relación según el tipo asignado al término almacenado en ella (diseño 2)

A continuación, se incluye un ejemplo en el que se puede observar cómo se van creando los nodos y relaciones del grafo, en varias iteraciones:

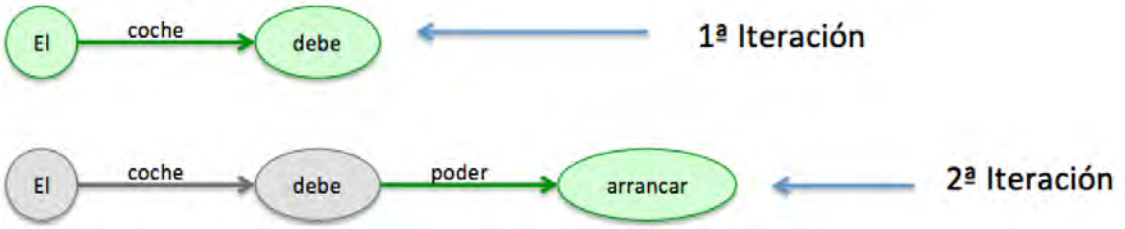


Figura 109: Ejemplo de funcionamiento del algoritmo de creación del segundo diseño

Como se puede observar en la *Figura 109*, en la primera iteración, se indexan tres términos del requisito ya que las relaciones también almacenan términos. Como en la primera iteración se han insertado tres términos (posiciones 0, 1 y 2 del array) y el bucle que recorre los términos de los requisitos, salta de cuatro en cuatro términos, en la segunda iteración se accedería directamente a la posición número 4 del array, es decir, al quinto término del requisito, saltándose la posición 3 (cuarto término). Este término no se pierde, ya que en la segunda iteración (en este caso) se incluye la relación que almacena dicho término. Básicamente, en cada iteración lo primero y lo último que se genera es un nodo, de modo que después de una iteración, queda pendiente en la siguiente crear la relación que una el último nodo creado en la iteración anterior con el creado en la nueva. En la implementación de este segundo diseño toma especial relevancia ya que las relaciones almacenan términos y si no se hubiera tenido en cuenta este detalle, se perderían términos.

5.1.3 IMPLEMENTACIÓN DE LA SOLUCIÓN FINAL

La implementación correspondiente a la creación del grafo y la indexación de los datos, con el diseño final, se realiza mediante los siguientes métodos:

- **ArrayList<String> readFile(String filePath):** este método recibe como parámetro la ruta de un fichero, el cual se lee y se devuelve, en un ArrayList, cada una de las líneas del mismo (cada línea en una posición del ArrayList). Este método se utiliza en *createDb(String dataPath, String patternsPath)* para leer el fichero que contiene los patrones.

- **void createDb(String dataPath, String patternsPath):** en la solución final, este método recibe dos parámetros; el primero de ellos se corresponde con la ruta del Excel en el que se encuentran los requisitos y el segundo, en el que se indica la ruta del fichero en

el que están los patrones con los que se indexarán los requisitos en el grafo. Los principales pasos que se realizan en este método son los siguientes:

1. En primer lugar, se recuperan todos los datos del Excel a través del API, para Java para documentos Microsoft, **Apache POI**, como se indicó en la implementación inicial.

2. En segundo lugar, se lee el fichero que contiene los patrones mediante el método *readFile(String filePath),* almacenando en cada posición de un ArrayList, cada uno de los patrones.

3. Seguidamente, se recorre el ArrayList en el que están los requisitos y por cada uno de ellos, se selecciona el patrón que lo generará y se separa el identificador del texto, como se indicó en la explicación de este mismo método de la implementación inicial.

4. Dentro del bucle del punto anterior, que recorre los requisitos, se incluye un bucle anidado en el que se recorre el array que contiene los términos del requisito que corresponda (según la iteración) para indexarlos en el grafo. Este proceso de indexación se realiza comparando, posición por posición, los términos que componen el requisito y los que conforman el patrón.

Más concretamente, el proceso de indexación se realiza recorriendo el array de términos, del requisito que corresponda, de uno en uno, de modo que se va comparando posición por posición con los términos del patrón, almacenando los términos en nodos o relaciones, según indique el patrón. Si el término del patrón es fijo (constante) pero coincide con el término del requisito, se indexa correctamente en un nodo, si el término del patrón estaba marcado con una "(n)", o en una relación, si estaba marcado con una "(r)". Si el término es variable, se indexa en un nodo o relación, según corresponda. Para ver esto más a fondo, se aconseja consultar el capítulo "Diseño de la solución final". Además, cabe resaltar que en esta implementación final, a diferencia de la anterior, todas las relaciones tienen el mismo tipo de relación, ya que la inclusión de más de un tipo es irrelevante para la realización del presente estudio. Por lo que el tipo de relación utilizado es "ESENCIAL".

Un aspecto importante a tener en cuenta en la implementación de este diseño final es cómo saber qué patrón va a generar qué requisitos. Para ello, se ha tomado la decisión de ordenar el Excel donde están los requisitos, de forma que todos los requisitos que se generan por un mismo patrón estén juntos. De esta forma, se tendrán primero todos los requisitos que se generan mediante el primer patrón del fichero de patrones, después todos los requisitos que se generan mediante el segundo patrón del fichero de patrones y así sucesivamente hasta completar todos los requisitos. Por lo que, de una ejecución se indexan todos los requisitos, cada uno mediante el patrón que le corresponda.

A continuación, se muestra una figura en la que se puede comprobar, iteración a iteración, cómo se indexaría el requisito "El coche debe poder arrancar" con el patrón "(n)El (n)coche (r)debe (r)poder (n)arrancar" con la implementación definitiva:

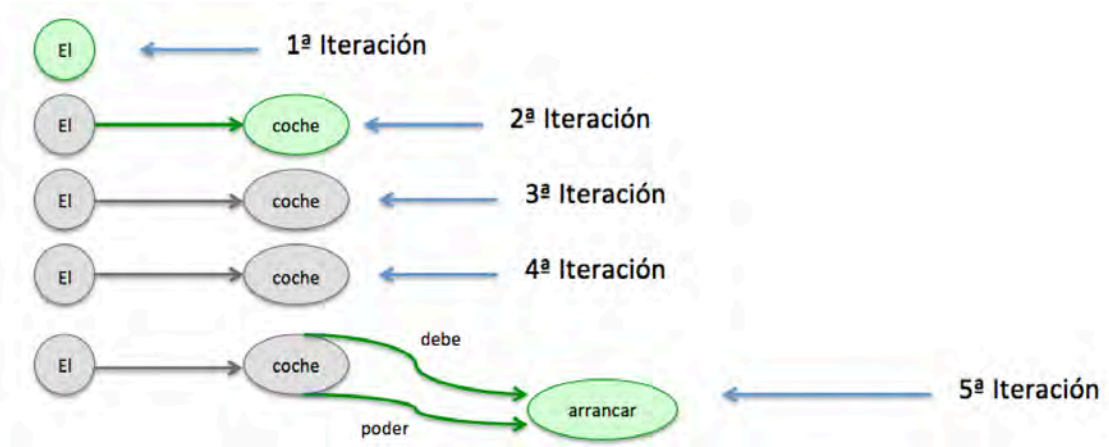


Figura 110: Ejemplo de funcionamiento del algoritmo de creación de la solución final

Como se puede ver en la *Figura 110*, en la primera iteración se crea el nodo "El". Como los términos del requisito y del patrón se van recorriendo uno a uno, en esa primera iteración no se efectúa ninguna otra acción. En la segunda iteración, se crea el nodo "coche", tal y como se indica en el patrón. Sin embargo, en la tercera iteración, no se modifica el grafo existente. Esto es debido a que tocaba crearse una relación, pero como se recorre uno a uno, hasta que el patrón no indique que se cree un nodo, esa relación no se insertará en el grafo. En la cuarta iteración sucede exactamente lo mismo que en la tercera. Los términos que se almacenarán en relaciones se van acumulando en un ArrayList hasta llegar a la creación de un nodo. Finalmente, el patrón indica que se cree el nodo "arrancar", por lo que se crean las relaciones acumuladas "debe" y "poder" y el propio nodo.

5.2 GENERACIÓN AUTOMÁTICA DE CONSULTAS

En este apartado se va a detallar el método utilizado para la generación automática de consultas. En primer lugar se explicará la implementación para el diseño inicial, para continuar después con la implementación que se realizó para el segundo diseño, que también sirve para la solución definitiva.

5.2.1 IMPLEMENTACIÓN INICIAL

En un primer momento, cuando el diseño que se estaba utilizando era el inicial, la generación de consultas era más sencilla, ya que sólo se almacenaban términos en los nodos y con un solo método se lograba generar los dos ficheros con las consultas. Ese método era **void getQueries(GraphDatabaseService graphDb, int numConsultas, int termMin, int termMax).** Los parámetros que recibía eran el grafo, el número de consultas que se querían generar y el número mínimo y máximo de términos por consulta.

Los principales pasos que seguía dicho método eran los siguientes: primeramente, se recuperaba el vocabulario del grafo, realizando sólo una consulta a los nodos recuperando así todos los términos. Seguidamente, en un bucle con tantas iteraciones como consultas había que generar, se seleccionaba de forma aleatoria el número de términos por consulta (teniendo en cuenta los parámetros *termMin* y *termMax*) y, en un bucle anidado con tantas iteraciones como términos fuese a tener la consulta, se seleccionaba un término que no hubiese sido escogido hasta el momento y se iban creando las consultas, tanto para Neo4j como KnowledgeMANAGER. Por ejemplo, si se querían cuatro consultas de entre 3 y 5 términos, se ejecutaba el método con esos parámetros *getQueries(graphDb, 4, 3, 5)* y se obtenían cuatro

consultas en Cypher para Neo4j y esas mismas consultas pero en lenguaje natural para KnowledgeMANAGER.

5.2.2 IMPLEMENTACIÓN DE LA SOLUCIÓN FINAL

A continuación se van a detallar los métodos utilizados, en la solución final, para generar las consultas tanto en Cypher (para ejecutar en Neo4j) como en lenguaje natural (para ejecutar en KnowledgeMANAGER):

- **void generateRelationships(String patternsPath, String dataPath):** como se puede apreciar, respecto de la implementación de este mismo método para la creación del grafo en el segundo diseño, este método recibe un parámetro más en el que se indica la ruta del fichero que contiene los patrones. Como con este diseño son los patrones los que marcan si un término debe almacenarse en un nodo o en relaciones, el propósito de este método se reduce a recopilar todos los términos que serán relaciones y escribirlos en el fichero *relacionesGrafo.txt*, cuya ruta será uno de los parámetros del método *getQueries* para consultarlo a la hora de generar las consultas.

Para ello, en primer lugar se lee el fichero que contiene los patrones mediante el método *readFile(String filePath)*. Después, se recorren los requisitos comparando cada uno de ellos con el patrón que lo genera, posición por posición, de modo que se van almacenando las relaciones en un ArrayList. Posteriormente, se escriben los términos, almacenados en ese ArrayList, en el fichero indicado anteriormente. En este caso, como en la creación del grafo sólo se utiliza el tipo de relación "ESENCIAL", se ha decidido no asignar ningún tipo de relación a los términos escritos en el fichero, como se hacía en la implementación anterior, ya que su inclusión era totalmente irrelevante en el presente estudio. Por lo que, en este caso, se escriben sólo los términos sin incluir los tipos de relación "ESENCIAL", "RECOMENDABLE" ni "OPCIONAL".

- **boolean searchTermInFile(String filePath, String term):** como ya se indicó anteriormente, este método recibe por parámetro la ruta de un fichero y un término y comprueba si dicho término se encuentra en el fichero. En este caso, el método devuelve *true* en caso de encontrar el término y *false* en caso opuesto. Se utilizará en el método *getQueries* para saber si un término está en el fichero *relacionesGrafo.txt* o no, es decir, para saber si un término está almacenado en un nodo o en relaciones.

- **void getQueries(GraphDatabaseService graphDb, int numConsultas, int termMin, int termMax, String relationshipsPath):** este es el principal método a la hora de generar las consultas. Recibe cinco parámetros, el primero de ellos se corresponde con el grafo del que se extraerá el vocabulario, en el segundo se indica el número de consultas que se quieren generar, el tercer y cuarto parámetro son el número de términos mínimo y máximo, respectivamente, entre los que oscilarán las consultas. Por su parte, en el último parámetro se incluye la ruta del fichero generado con el método *generateRelationships* que incluye los términos que están almacenados en relaciones.

Cabe destacar que este método genera dos ficheros, uno en el que se escriben las consultas en Cypher y otro en el que se escriben las consultas en lenguaje natural. A continuación, se van a detallar los principales pasos que se realizan para lograr generar dichas consultas:

1. En primer lugar se recupera el vocabulario del grafo, es decir, todos los términos de los requisitos almacenados en el mismo. Para ello, se realizan las consultas que se indicaron en las *figuras 94* y *95*, y se almacenan todos los términos en un ArrayList.

2. Seguidamente, se procede a generar las consultas, teniendo en cuenta los datos recibidos por parámetro. Se crea un bucle en el que se generarán el número de consultas indicadas en el segundo parámetro. Dentro de ese bucle, lo primero que se hace es generar, de forma aleatoria, el número de términos que tendrá la consulta. Para ello, se genera un número aleatorio que esté entre el tercer y cuarto parámetro (ambos incluidos).

3. Dentro del bucle del punto anterior, se crea otro bucle anidado en el que se realizarán tantas iteraciones como términos vaya a tener la consulta. En este nuevo bucle, primero se selecciona un término del vocabulario, que no haya sido seleccionado todavía. Después, se comienzan a generar tanto la consulta para Neo4j como para KnowledgeMANAGER. Para ello, mediante el método searchTermInFile, se consulta si el término seleccionado está almacenado en relaciones o en un nodo. Si está almacenado en nodos, para Neo4j, se crea una consulta del tipo "Match (n)", mientras que si está en relaciones se crea una consulta del tipo "Match ()-[r]->()". De modo que, si para una consulta de dos términos, los dos están almacenados en nodos, se genera sólo una consulta del primer tipo; si los dos están almacenados en relaciones, también se crea sólo una consulta pero del segundo tipo; por último, en el caso de que un término estuviese en un nodo y el otro en relaciones, se crearían dos consultas de la forma que se detalló en la *Figura 96*.

4. Una vez que se hayan seleccionado todos los términos de la consulta, se hayan formado las dos consultas y se hayan escrito en los ficheros correspondientes (*consultasGeneradasNeo4j.txt* y *consultasGeneradasKM.txt*), se vuelve a repetir el proceso tantas veces como consultas se vayan a generar (primer bucle). A continuación se incluyen dos imágenes en las que se puede observar cómo quedan ambos ficheros después de ejecutar getQueries(grapDb, 6, 1, 3, "/……./relacionesGrafo.txt"):

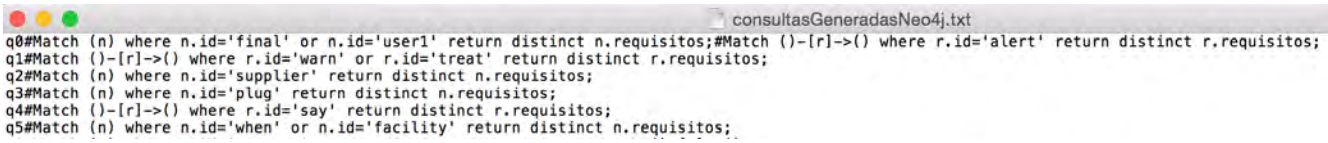


Figura 111: Fichero con las consultas en Cypher

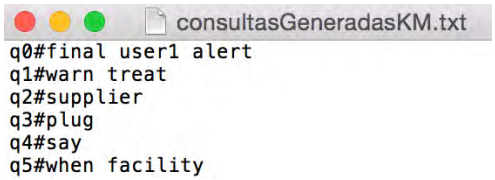


Figura 112: Fichero con las consultas en lenguaje natural

A partir de las *Figuras 111* y *112*, se comprueba cómo se escriben las consultas en ambos ficheros. En ambos casos, cada línea se comienza con el identificador de la consulta seguido de una almohadilla (#). En el caso de las consultas en Cypher, como se ha comentado anteriormente, hay líneas que contienen dos consultas, una para nodos y otra para relaciones, separadas por una almohadilla. Como se puede observar en la *Figura 112*, las consultas para KnowledgeMANAGER simplemente constan de los términos seleccionados. Se han incluido las almohadillas en ambos ficheros para que el procesamiento de las consultas sea más sencillo.

5.3 EJECUCIÓN AUTOMÁTICA DE CONSULTAS EN NEO4J

Para la ejecución automática de consultas en Neo4j, se ha implementado el método **void runQueries(GraphDatabaseService graphDb, String filePath)**, en el que se reciben dos parámetros, el primero se corresponde con el grafo sobre el que se ejecutarán las consultas y en el segundo se indica la ruta del fichero *consultasGeneradasNeo4j.txt*, creado en el proceso de generación de consultas.

En este método, principalmente, se realizan los siguientes pasos:

1. Se comienza creando un bucle con tantas iteraciones como líneas con consultas tenga el fichero. A cada iteración, lo primero que se hace es separar el identificador de la consulta, "q0" por ejemplo, de la consulta mediante el método Split("#"). En el caso de que la línea contenga dos consultas (una para nodos y otra para relaciones) también se separan ya que como se comprobaba en la *Figura 111*, las consultas se separaban también por #.

2. Seguidamente, también en cada iteración, se ejecuta la consulta y se recorren los resultados de la misma, almacenando los identificadores devueltos en una variable de String. Cabe destacar que si un requisito se devuelve más de una vez, sólo se escribe en la variable una única vez.

3. Este punto es opcional. Sólo en el caso de que hubiesen dos consultas en una línea, como ocurre en la consulta "q0" de la *Figura 111*, se ejecuta la segunda de ellas, acumulando el resto de identificadores de requisitos devueltos en la misma variable que en el punto dos.

4. Finalmente, se escribe la variable de String, que contiene todos los identificadores de requisitos devueltos, en un fichero denominado *resultadosConsultasNeo4j.txt*.

Todos estos pasos se repiten a cada iteración, dando como resultado el fichero comentado en el punto 4, con todos los resultados de todas las consultas que había para ejecutar en el fichero *consultasGeneradasNeo4j.txt*. A continuación, se incluye una imagen en la que se puede comprobar cómo queda el fichero *resultadosConsultasNeo4j.txt* después de una ejecución cualquiera:

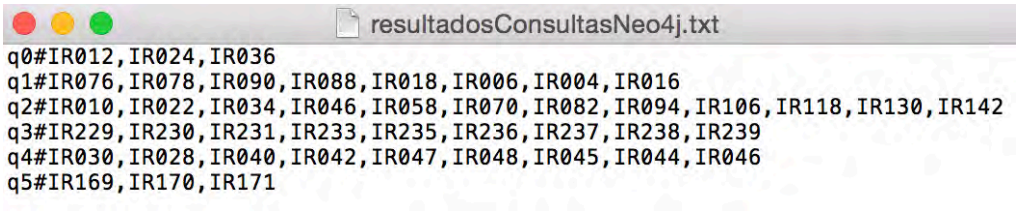


Figura 113: Formato del fichero resultadosConsultasNeo4j.txt

Como se puede comprobar en la *Figura 113*, los resultados de ejecutar las consultas se incluyen en el fichero indicado. Se incluye el identificador de cada consulta, seguido de una almohadilla y de los resultados correspondientes a cada una de ellas.

6. EXPERIMENTACIÓN

En este capítulo se va a detallar el experimento realizado, presentando los resultados obtenidos y analizándolos de forma exhaustiva.

6.1 ASPECTOS RELEVANTES

6.1.1 DISEÑO DEL EXPERIMENTO

En este apartado se indican todos y cada uno de los pasos identificados para realizar el experimento. Se detallan en orden cronológico, indicando brevemente en qué consiste cada uno de ellos.

1. **Definir el conjunto de datos a utilizar:** en primer lugar hay que decidir qué data set se va a utilizar en el experimento. En este caso, se ha tomado una especificación de requisitos (1.572 requisitos).
2. **Crear el grafo en Neo4j:** una vez se tienen los datos a indexar en ambos sistemas, se crea el grafo en Neo4j indexando los requisitos tanto en nodos como en relaciones.
3. **Indexar los requisitos en KnowledgeMANAGER**.
4. **Definir un conjunto de consultas:** una vez se han indexado los requisitos en ambos sistemas, se define el conjunto de consultas que se ejecutarán. En este caso, se han generado las consultas a través del proceso automatizado desarrollado para este trabajo. En este caso, se ha decidido incluir en cada consulta entre 1 y 3 términos, ya que muchas de las consultas que se ejecutan normalmente en los buscadores, suelen estar compuestas en esa franja de términos. Cabe destacar que un término, se refiere a cada una de las palabras que consta cada requisito, en este experimento.
5. **Identificación de los resultados relevantes a cada consulta.** Para este experimento, sólo se van a tener en cuenta los 20 primeros resultados devueltos por cada sistema. Se ha tomado esta decisión, ya que es muy complicado que el usuario llegue hasta el resultado número 21 o más allá, para una consulta que haya realizado. Si no encuentra resultados relevantes entre los 10-20 primeros resultados, seguramente modifique la consulta, en lugar de seguir navegando más allá del resultado 20. Por lo que, todos aquellos resultados relevantes que sean devueltos después del resultado número 20, no serán tenidos en cuenta.
6. **Ejecutar las consultas tanto en Neo4j como en KnowledgeMANAGER.**
7. **Calcular la precisión y recall arrojada por cada sistema para cada consulta.**
8. **Analizar cuál de los dos sistemas ofrece mejores resultados para el experimento.** Dicho análisis se apoya en distintos gráficos que ayudan a la lectura de los resultados.

6.1.2 ¿EN QUÉ CONSISTE EL EXPERIMENTO?

Como ya se ha comentado en capítulos anteriores, el **presente estudio** trata de comparar **cómo de bien recuperan resultados relevantes RSHP y Neo4j a consultas sobre términos en concreto**. Resultados relevantes, para este estudio, serán aquellos requisitos que contengan, al menos, uno de los términos de la consulta. En este apartado se va a comentar el experimento realizado, desde la indexación de los datos en ambos sistemas hasta la ejecución de las consultas generadas y el análisis de los resultados obtenidos.

En primer lugar, cabe destacar que los datos que se han utilizado para este estudio, son un conjunto de requisitos, concretamente, **1.572 requisitos** escritos en inglés, que indican

diferentes acciones a realizar por distintos usuarios ante diferentes situaciones. A continuación se muestra una imagen del Excel que contiene los datos:

	A	B
1	ID	Description
2	IR001	The client shall be able to treat system failures
3	IR002	The user shall be able to treat system failures
4	IR003	The author shall be able to treat system failures
5	IR005	The customer shall be able to treat system failures
6	IR007	The staff shall be able to treat system failures
7	IR008	The administrator shall be able to treat system failures
8	IR009	The stakeholder shall be able to treat system failures
9	IR010	The supplier shall be able to treat system failures
10	IR011	The commission shall be able to treat system failures
11	IR012	The user1 shall be able to treat system failures
12	IR013	The client must be able to treat system failures
13	IR014	The user must be able to treat system failures
14	IR015	The author must be able to treat system failures
15	IR017	The customer must be able to treat system failures
16	IR019	The staff must be able to treat system failures
17	IR020	The administrator must be able to treat system failures
18	IR021	The stakeholder must be able to treat system failures
19	IR022	The supplier must be able to treat system failures
20	IR023	The commission must be able to treat system failures
21	IR024	The user1 must be able to treat system failures

Figura 114: Muestra de los requisitos utilizados para el estudio

Como se puede observar en la *Figura 114*, el Excel utilizado sólo contiene dos columnas, una para el id de cada requisito y otra para indicar el texto de cada uno de ellos. Seguidamente, se van a detallar los aspectos más relevantes del proceso de indexación en Neo4j.

En Neo4j, se han indexado los datos de la forma que se detalló en los capítulos de *Análisis, Diseño* e *Implementación* de la solución final. Básicamente, se crea el grafo indexando los datos de los requisitos, a través de patrones, tanto en los nodos como en las relaciones. Los patrones utilizados para indexar los 1.572 requisitos en Neo4j han sido los siguientes:

1- (n)The (n)[user] (r)[shall] (n)be (r)able (n)to (r)[treat] (n)system (n)failures

2- (n)The (n)[final] (n)user (r)[shall] (n)be (r)able (n)to (r)[treat] (n)system (n)failures

3- (n)When (n)the (n)incident (r)is (r)generated (n)the (n)[client] (r)shall (n)[connect] (n)1 (n)[functionality] (r)per (n)[second]

4- (n)When (n)the (n)incident (r)is (r)generated (n)the (n)[final] (n)user (r)shall (n)[connect] (n)1 (n)[functionality] (r)per (n)[second]

5- (n)When (n)the (n)incident (r)is (r)generated (n)the (n)[client] (r)shall (n)[assemble] (n)1 (n)functional (r)[unit] (r)per (n)[second]

6- (n)When (n)the (n)incident (r)is (r)generated (n)the (n)[final] (n)user (r)shall (n)[assemble] (n)1 (n)functional (r)[unit] (r)per (n)[second]

7- (n)When (n)the (n)incident (r)is (r)generated (n)the (n)[staff] (r)shall (n)[join] (n)1 (n)functional (r)[simulation] (n)[facility] (r)per (n)[second]

8- (n)When (n)the (n)incident (r)is (r)generated (n)the (n)[end] (n)user (r)shall (n)[join] (n)1 (n)functional (r)[simulation] (n)[facility] (r)per (n)[second]

Figura 115: Patrones para la indexación en Neo4j

En el caso de KnowledgeMANAGER, cada requisito se ha incluido en un fichero distinto, creando de esta forma 1.572 ficheros almacenados en una carpeta. Esta carpeta es la que se ha indexado en KM, en la pantalla que se observa en la siguiente figura:

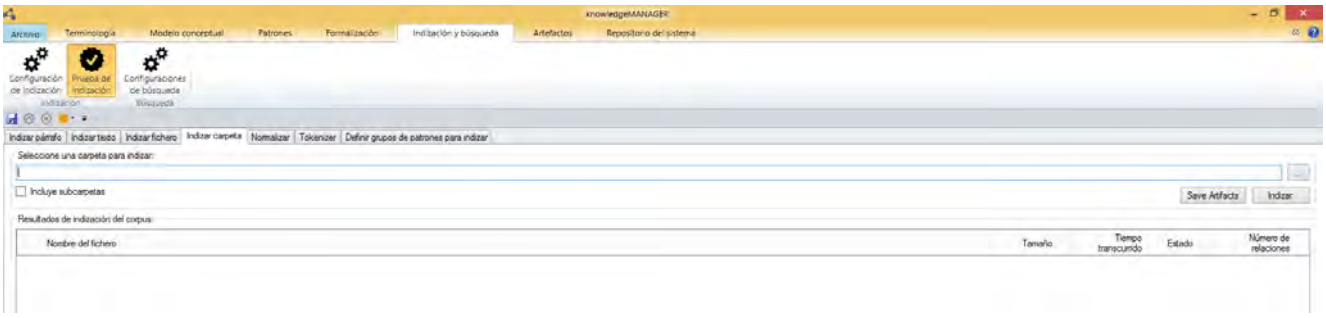


Figura 116: Pantalla indexar carpeta en KM

Después de indexar los datos, se debe hacer click en el botón “Save Artifacts” que se observa en la imagen. Con esto, se crea un artefacto por cada uno de los requisitos indexados. Cabe destacar, que un artefacto contiene los términos con relevancia semántica de un requisito. De esta forma, por cada uno de los artefactos, KM crea una representación formal que incluye cada uno de esos términos con importancia semántica, como se puede comprobar con la siguiente figura:

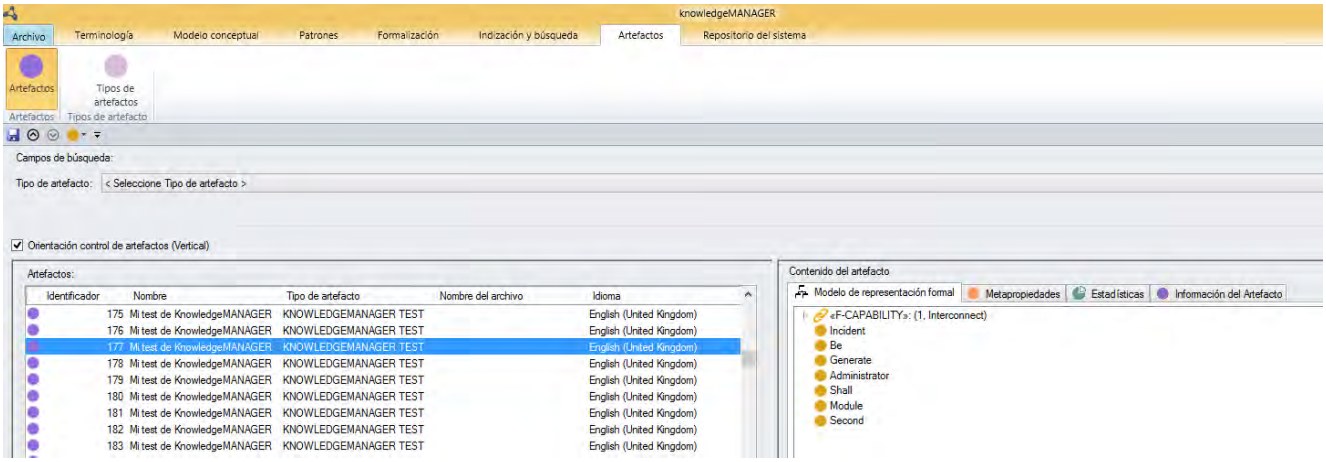


Figura 117: Artefactos en KM

En la *Figura 117* se observan algunos de los artefactos creados. Como se puede comprobar, por cada artefacto existen una serie de pestañas con información del mismo (modelo de representación formal, meta-propiedades, estadísticas e información del artefacto). En este caso, hay seleccionado uno de ellos que se corresponde con el requisito "When the incident is generated the administrator shall interconnect 1 module per second". Se puede comprobar como las palabras con una semántica baja (when, the, per), se eliminan de la representación formal del artefacto. Esto, como se verá algo más adelante, afectará negativamente a las consultas que incluyan alguno de esos términos no presentes en la representación formal.

Después de tener los datos indexados en ambos sistemas, se han generado 30 consultas de entre 1 y 3 términos (ambos incluidos) de forma aleatoria, utilizando el mecanismo descrito en los capítulos de *Análisis*, *Diseño* e *Implementación* de la solución final. Cada una de las consultas han sido generadas tanto en lenguaje natural (para RSHP) como en Cypher (para Neo4j). En el apartado *Resultados Obtenidos* se indican las consultas con los resultados obtenidos en cada sistema.

La ejecución de las consultas en Neo4j se ha realizado de forma programática, como se explicó en los capítulos indicados. No obstante, se ha utilizado la interfaz de navegador de Neo4j para comprobar que los grafos se generaban de forma correcta.

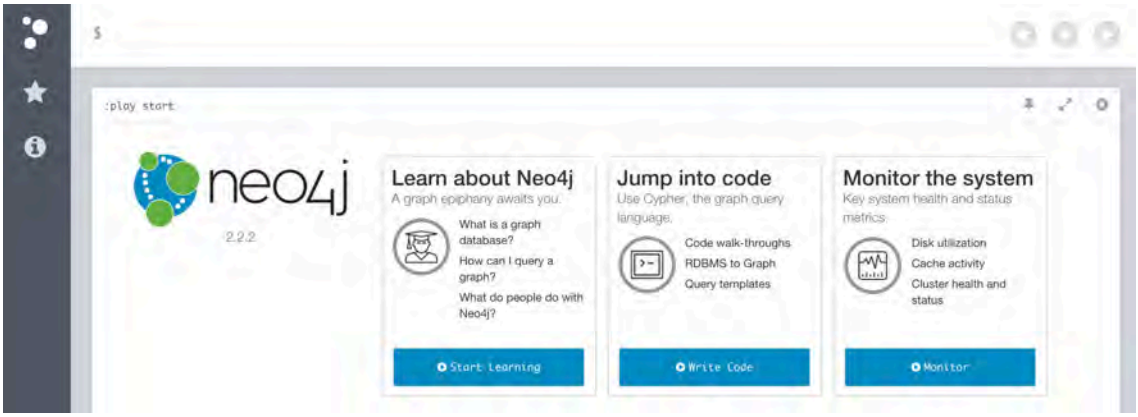


Figura 118: Interfaz de navegador de Neo4j

En cuanto a la ejecución de las consultas en RSHP, cabe resaltar que se puede efectuar de dos formas distintas. **KM** permite realizar búsquedas por inclusión y por similitud. La

búsqueda por inclusión, como su propio nombre indica, recupera artefactos que incluyan los términos de la consulta o términos que guarden algún tipo de relación con los de la consulta. La búsqueda por similitud, tiene en cuenta el número de veces que aparecen los términos en los artefactos. Es decir, se calculan los valores para el tf e idf y en función de los resultados, se decide si un artefacto es relevante o no para la consulta. Para este experimento, se ha utilizado la **búsqueda por inclusión**, ya que lo que se pretende es recuperar artefactos que incluyan términos de la consulta. A continuación se indica la pantalla en la que se han ejecutado las consultas en KM:

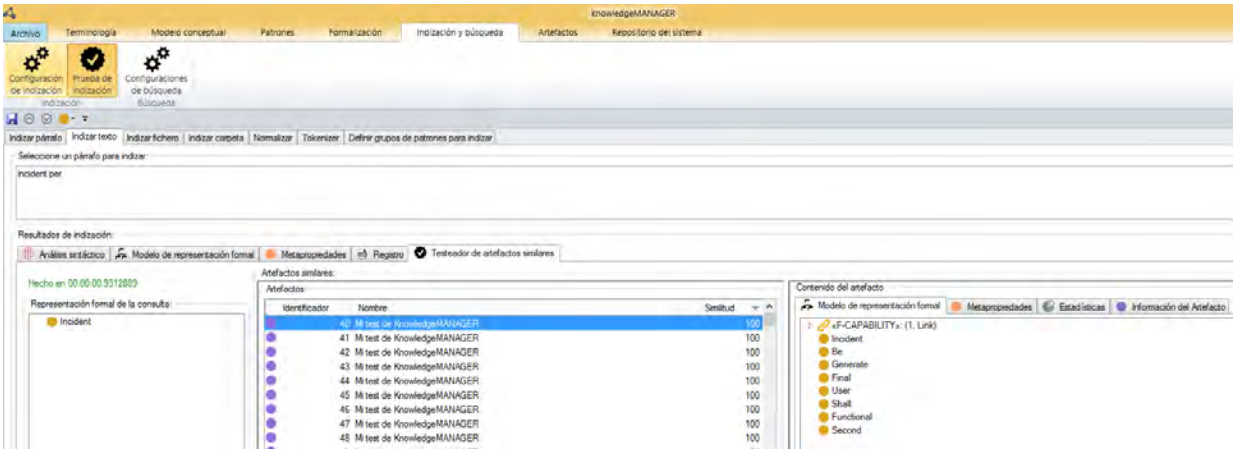


Figura 119: Pantalla ejecución consultas en KM

Cuando se ejecutan las consultas en KM, los resultados obtenidos son los artefactos, que corresponden a cada uno de los requisitos. En la *Figura 119*, se puede comprobar como para una consulta dada, KM realiza la representación formal de la consulta, recupera los artefactos similares y por cada uno de los artefactos permite acceder a su contenido, a través del contenedor de la derecha denominado “Contenido del artefacto”. Además, como se ha comentado anteriormente, en la representación formal de la consulta se elimina el término “per” ya que no tiene suficiente semántica. Esto provoca que sólo se busque por el término “incident”, lo que puede conllevar que no se recuperen todos los artefactos relevantes para este estudio.

Cabe destacar, que para el estudio realizado, **sólo se van a tener en cuenta los 20 primeros resultados devueltos**, por cada uno de los sistemas, para cada consulta. Es decir, se va a realizar un cut-off a los 20 primeros resultados. Además, cabe recordar que el objetivo de este estudio es conocer cuál de los dos sistemas recupera mejor requisitos que contienen algún término de la consulta.

6.1.3 MEDIDAS DE PRECISIÓN, RECALL Y F-MEASURE

Para cada una de las consultas, se va a calcular la precisión y la recall para cada sistema. Por ello, en este apartado se van a indicar las fórmulas con las que se van a calcular dichas medidas.

La **precisión** mide el ruido de los resultados de una consulta, es decir, la cantidad de resultados no relevantes recuperados. Esta medida se va a calcular mediante la siguiente fórmula:

$$Precision = \frac{\#(documentos\ relevantes\ recuperados)}{\#(documentos\ recuperados)} = \frac{|recuperados \cap relevantes|}{|recuperados|}$$

Figura 120: Fórmula para calcular la precisión

Como se puede comprobar en la *Figura 120*, para calcular la precisión se van a dividir los requisitos relevantes recuperados entre el total de recuperados. En este caso, si se devuelven más de 20 requisitos, se dividirá entre 20 ya que, como se ha comentado, se va a realizar un cut-off a los 20 primeros resultados.

Por su parte, la **recall** mide la exhaustividad, es decir, mide el número de resultados relevantes recuperados respecto del total de relevantes para una consulta. Esta medida se va a calcular a través de la siguiente fórmula:

$$Recall = \frac{\#(documentos\ relevantes\ recuperados)}{\#(documentos\ relevantes)} = \frac{|recuperados \cap relevantes|}{|relevantes|}$$

Figura 121: Fórmula para calcular la recall

A partir de la *Figura 121*, se comprueba como la recall se calcula dividiendo los requisitos relevantes recuperados entre el total de relevantes para cada consulta.

La **medida F-measure**, también conocida como F_1 score, agrupa las medidas de precisión y recall en una sola. Es sensible a diferencias grandes. La fórmula utilizada para su cálculo es la siguiente:

$$F = \frac{2}{\frac{1}{P} + \frac{1}{R}} = \frac{2 \cdot P \cdot R}{P + R}$$

Figura 122: Fórmula para calcular F-measure

6.2 RESULTADOS OBTENIDOS

En este apartado se van a presentar los resultados obtenidos después de ejecutar las mismas 30 consultas en KM y Neo4j. Cabe destacar que los resultados para cada consulta se van a mostrar en tablas para facilitar su lectura. El modelo de tabla que se va a utilizar es el que se incluye a continuación:

Consulta X		
Número de términos de la consulta	X	
Requisitos relevantes para la consulta	XXX	
Sistema	**RSHP**	**Neo4j**
Sintaxis consulta	ejemplo	Match (n) where n.id=’ejemplo’ return distinct n.requisitos;
Total requisitos recuperados	XX	XX
Relevantes de los 20 primeros recuperados	XX	XX
Precisión	X	X
Recall	X	X

Tabla 20: Modelo de tabla para presentar los resultados del estudio

Como se puede observar en la *Tabla 20*, para cada consulta se va a indicar el número de términos que la componen, el número total de requisitos relevantes para la misma (cuántos son relevantes de los 1.572 requisitos totales), la sintaxis empleada para cada sistema, el número total de requisitos recuperados por cada sistema, el número de relevantes de los 20 primeros recuperados y la precisión y recall también para cada uno de los sistemas. Seguidamente, se presentan los resultados de las 30 consultas ejecutadas:

Consulta 1		
Número de términos de la consulta	3	
Requisitos relevantes para la consulta	282	
Sistema	**RSHP**	**Neo4j**
Sintaxis consulta	final user1 alert	Match (n) where n.id='final' or n.id='user1' return distinct n.requisitos; Match ()-[r]->() where r.id='alert' return distinct r.requisitos;
Total requisitos recuperados	8	282
Relevantes de los 20 primeros recuperados	8	20
Precisión	1	1
Recall	0,03	0,07

Tabla 21: Resultados consulta 1

Consulta 2		
Número de términos de la consulta	2	
Requisitos relevantes para la consulta	48	
Sistema	**RSHP**	**Neo4j**
Sintaxis consulta	warn treat	Match ()-[r]->() where r.id='warn' or r.id='treat' return distinct r.requisitos;
Total requisitos recuperados	72	48
Relevantes de los 20 primeros recuperados	5	20
Precisión	0,25	1
Recall	0,1	0,42

Tabla 22: Resultados consulta 2

Consulta 3		
Número de términos de la consulta	1	
Requisitos relevantes para la consulta	131	
Sistema	**RSHP**	**Neo4j**
Sintaxis consulta	supplier	Match (n) where n.id='supplier' return distinct n.requisitos;
Total requisitos recuperados	131	100
Relevantes de los 20 primeros recuperados	20	16
Precisión	1	0,8
Recall	0,15	0,12

Tabla 23: Resultados consulta 3

Consulta 4		
Número de términos de la consulta	1	
Requisitos relevantes para la consulta	156	
Sistema	**RSHP**	**Neo4j**
Sintaxis consulta	plug	Match (n) where n.id='plug' return distinct n.requisitos;
Total requisitos recuperados	0	156
Relevantes de los 20 primeros recuperados	0	20
Precisión	0	1
Recall	0	0,13

Tabla 24: Resultados consulta 4

Consulta 5		
Número de términos de la consulta	1	
Requisitos relevantes para la consulta	24	
Sistema	**RSHP**	**Neo4j**
Sintaxis consulta	say	Match ()-[r]->() where r.id='say' return distinct r.requisitos;
Total requisitos recuperados	48	24
Relevantes de los 20 primeros recuperados	0	20
Precisión	0	1
Recall	0	0,83

Tabla 25: Resultados consulta 5

Consulta 6		
Número de términos de la consulta	2	
Requisitos relevantes para la consulta	1.404	
Sistema	**RSHP**	**Neo4j**
Sintaxis consulta	when facility	Match (n) where n.id='when' or n.id='facility' return distinct n.requisitos;
Total requisitos recuperados	216	1.404
Relevantes de los 20 primeros recuperados	20	20
Precisión	1	1
Recall	0,01	0,01

Tabla 26: Resultados consulta 6

Consulta 7		
Número de términos de la consulta	2	
Requisitos relevantes para la consulta	1.404	
Sistema	**RSHP**	**Neo4j**
Sintaxis consulta	manager per	Match (n) where n.id='manager' return distinct n.requisitos; Match ()-[r]->() where r.id='per' return distinct r.requisitos;
Total requisitos recuperados	230	1.404
Relevantes de los 20 primeros recuperados	18	20
Precisión	0,9	1
Recall	0,01	0,01

Tabla 27: Resultados consulta 7

Consulta 8		
Número de términos de la consulta	3	
Requisitos relevantes para la consulta	1.460	
Sistema	**RSHP**	**Neo4j**
Sintaxis consulta	stakeholder incident user	Match (n) where n.id='stakeholder' or n.id='incident' or n.id='user' return distinct n.requisitos;
Total requisitos recuperados	718	1.114
Relevantes de los 20 primeros recuperados	20	15
Precisión	1	0,75
Recall	0,01	0,01

Tabla 28: Resultados consulta 8

Consulta 9		
Número de términos de la consulta	2	
Requisitos relevantes para la consulta	1.572	
Sistema	**RSHP**	**Neo4j**
Sintaxis consulta	interrelate the	Match (n) where n.id='interrelate' or n.id='the' return distinct n.requisitos;
Total requisitos recuperados	0	1.572
Relevantes de los 20 primeros recuperados	0	20
Precisión	0	1
Recall	0	0,01

Tabla 29: Resultados consulta 9

Consulta 10		
Número de términos de la consulta	2	
Requisitos relevantes para la consulta	444	
Sistema	**RSHP**	**Neo4j**
Sintaxis consulta	unit assemble	Match (n) where n.id='assemble' return distinct n.requisitos; Match ()-[r]->() where r.id='unit' return distinct r.requisitos;
Total requisitos recuperados	324	444
Relevantes de los 20 primeros recuperados	20	20
Precisión	1	1
Recall	0,05	0,05

Tabla 30: Resultados consulta 10

Consulta 11		
Número de términos de la consulta	2	
Requisitos relevantes para la consulta	1.164	
Sistema	**RSHP**	**Neo4j**
Sintaxis consulta	must second	Match (n) where n.id='second' return distinct n.requisitos; Match ()-[r]->() where r.id='must' return distinct r.requisitos;
Total requisitos recuperados	1.164	1.164
Relevantes de los 20 primeros recuperados	20	20
Precisión	1	1
Recall	0,02	0,02

Tabla 31: Resultados consulta 11

Consulta 12		
Número de términos de la consulta	2	
Requisitos relevantes para la consulta	168	
Sistema	**RSHP**	**Neo4j**
Sintaxis consulta	talk to	Match (n) where n.id='to' return distinct n.requisitos; Match ()-[r]->() where r.id='talk' return distinct r.requisitos;
Total requisitos recuperados	12	168
Relevantes de los 20 primeros recuperados	12	20
Precisión	1	1
Recall	0,07	0,12

Tabla 32: Resultados consulta 12

Consulta 13		
Número de términos de la consulta	1	
Requisitos relevantes para la consulta	131	
Sistema	**RSHP**	**Neo4j**
Sintaxis consulta	author	Match (n) where n.id='author' return distinct n.requisitos;
Total requisitos recuperados	131	131
Relevantes de los 20 primeros recuperados	20	20
Precisión	1	1
Recall	0,15	0,15

Tabla 33: Resultados consulta 13

Consulta 14		
Número de términos de la consulta	1	
Requisitos relevantes para la consulta	168	
Sistema	**RSHP**	**Neo4j**
Sintaxis consulta	able	Match ()-[r]->() where r.id='able' return distinct r.requisitos;
Total requisitos recuperados	168	45
Relevantes de los 20 primeros recuperados	20	18
Precisión	1	0,9
Recall	0,12	0,11

Tabla 34: Resultados consulta 14

Consulta 15		
Número de términos de la consulta	1	
Requisitos relevantes para la consulta	156	
Sistema	**RSHP**	**Neo4j**
Sintaxis consulta	connect	Match (n) where n.id='connect' return distinct n.requisitos;
Total requisitos recuperados	0	156
Relevantes de los 20 primeros recuperados	0	20
Precisión	0	1
Recall	0	0,13

Tabla 35: Resultados consulta 15

Consulta 16		
Número de términos de la consulta	1	
Requisitos relevantes para la consulta	24	
Sistema	**RSHP**	**Neo4j**
Sintaxis consulta	communicate	Match ()-[r]->() where r.id='communicate' return distinct r.requisitos;
Total requisitos recuperados	48	24
Relevantes de los 20 primeros recuperados	8	20
Precisión	0,4	1
Recall	0,33	0,83

Tabla 36: Resultados consulta 16

Consulta 17		
Número de términos de la consulta	2	
Requisitos relevantes para la consulta	1.404	
Sistema	**RSHP**	**Neo4j**
Sintaxis consulta	generated file	Match ()-[r]->() where r.id='generated' or r.id='file' return distinct r.requisitos;
Total requisitos recuperados	1.404	1.404
Relevantes de los 20 primeros recuperados	20	20
Precisión	1	1
Recall	0,01	0,01

Tabla 37: Resultados consulta 17

Consulta 18		
Número de términos de la consulta	3	
Requisitos relevantes para la consulta	324	
Sistema	**RSHP**	**Neo4j**
Sintaxis consulta	system link inform	Match (n) where n.id='system' or n.id='link' return distinct n.requisitos; Match ()-[r]->() where r.id='inform' return distinct r.requisitos;
Total requisitos recuperados	24	324
Relevantes de los 20 primeros recuperados	20	20
Precisión	1	1
Recall	0,06	0,06

Tabla 38: Resultados consulta 18

Consulta 19		
Número de términos de la consulta	2	
Requisitos relevantes para la consulta	1.418	
Sistema	**RSHP**	**Neo4j**
Sintaxis consulta	1 commission	Match (n) where n.id='1' or n.id='commission' return distinct n.requisitos;
Total requisitos recuperados	1.418	1.228
Relevantes de los 20 primeros recuperados	20	16
Precisión	1	0,8
Recall	0,01	0,01

Tabla 39: Resultados consulta 19

Consulta 20		
Número de términos de la consulta	2	
Requisitos relevantes para la consulta	348	
Sistema	**RSHP**	**Neo4j**
Sintaxis consulta	interconnect functionality	Match (n) where n.id='interconnect' or n.id='functionality' return distinct n.requisitos;
Total requisitos recuperados	216	348
Relevantes de los 20 primeros recuperados	20	20
Precisión	1	1
Recall	0,06	0,06

Tabla 40: Resultados consulta 20

Consulta 21		
Número de términos de la consulta	2	
Requisitos relevantes para la consulta	216	
Sistema	**RSHP**	**Neo4j**
Sintaxis consulta	module numerical	Match (n) where n.id='module' return distinct n.requisitos; Match ()-[r]->() where r.id='numerical' return distinct r.requisitos;
Total requisitos recuperados	216	216
Relevantes de los 20 primeros recuperados	20	20
Precisión	1	1
Recall	0,09	0,09

Tabla 41: Resultados consulta 21

Consulta 22		
Número de términos de la consulta	1	
Requisitos relevantes para la consulta	156	
Sistema	**RSHP**	**Neo4j**
Sintaxis consulta	relate	Match (n) where n.id='relate' return distinct n.requisitos;
Total requisitos recuperados	0	134
Relevantes de los 20 primeros recuperados	0	18
Precisión	0	0,9
Recall	0	0,12

Tabla 42: Resultados consulta 22

Consulta 23		
Número de términos de la consulta	2	
Requisitos relevantes para la consulta	1.488	
Sistema	**RSHP**	**Neo4j**
Sintaxis consulta	functional shall	Match (n) where n.id='functional' return distinct n.requisitos; Match ()-[r]->() where r.id='shall' return distinct r.requisitos;
Total requisitos recuperados	1.488	1.488
Relevantes de los 20 primeros recuperados	20	20
Precisión	1	1
Recall	0,01	0,01

Tabla 43: Resultados consulta 23

Consulta 24		
Número de términos de la consulta	2	
Requisitos relevantes para la consulta	324	
Sistema	**RSHP**	**Neo4j**
Sintaxis consulta	simulator functionally	Match (n) where n.id='simulator' or n.id='functionally' return distinct n.requisitos;
Total requisitos recuperados	216	324
Relevantes de los 20 primeros recuperados	20	20
Precisión	1	1
Recall	0,06	0,06

Tabla 44: Resultados consulta 24

Consulta 25		
Número de términos de la consulta	1	
Requisitos relevantes para la consulta	131	
Sistema	**RSHP**	**Neo4j**
Sintaxis consulta	customer	Match (n) where n.id='customer' return distinct n.requisitos;
Total requisitos recuperados	131	131
Relevantes de los 20 primeros recuperados	20	20
Precisión	1	1
Recall	0,15	0,15

Tabla 45: Resultados consulta 25

Consulta 26		
Número de términos de la consulta	3	
Requisitos relevantes para la consulta	417	
Sistema	**RSHP**	**Neo4j**
Sintaxis consulta	plug1 join client	Match (n) where n.id='plug1' or n.id='join' or n.id='client' return distinct n.requisitos;
Total requisitos recuperados	13	346
Relevantes de los 20 primeros recuperados	13	16
Precisión	1	0,8
Recall	0,03	0,04

Tabla 46: Resultados consulta 26

Consulta 27		
Número de términos de la consulta	2	
Requisitos relevantes para la consulta	1.404	
Sistema	**RSHP**	**Neo4j**
Sintaxis consulta	is simulation	Match ()-[r]->() where r.id='is' or r.id='simulation' return distinct r.requisitos;
Total requisitos recuperados	1.572	1.404
Relevantes de los 20 primeros recuperados	20	20
Precisión	1	1
Recall	0,01	0,01

Tabla 47: Resultados consulta 27

Consulta 28		
Número de términos de la consulta	3	
Requisitos relevantes para la consulta	582	
Sistema	**RSHP**	**Neo4j**
Sintaxis consulta	minute be administrator	Match (n) where n.id='minute' or n.id='be' or n.id='administrator' return distinct n.requisitos;
Total requisitos recuperados	844	582
Relevantes de los 20 primeros recuperados	20	20
Precisión	1	1
Recall	0,03	0,03

Tabla 48: Resultados consulta 28

Consulta 29		
Número de términos de la consulta	3	
Requisitos relevantes para la consulta	384	
Sistema	**RSHP**	**Neo4j**
Sintaxis consulta	staff failures engineering	Match (n) where n.id='staff' or n.id='failures' return distinct n.requisitos; Match ()-[r]->() where r.id='engineering' return distinct r.requisitos;
Total requisitos recuperados	23	384
Relevantes de los 20 primeros recuperados	20	20
Precisión	1	1
Recall	0,05	0,05

Tabla 49: Resultados consulta 29

Consulta 30		
Número de términos de la consulta	2	
Requisitos relevantes para la consulta	153	
Sistema	**RSHP**	**Neo4j**
Sintaxis consulta	end warn	Match (n) where n.id='end' return distinct n.requisitos; Match ()-[r]->() where r.id='warn' return distinct r.requisitos;
Total requisitos recuperados	175	153
Relevantes de los 20 primeros recuperados	16	20
Precisión	0,8	1
Recall	0,1	0,13

Tabla 50: Resultados consulta 30

Para finalizar este apartado, se incluye una tabla en la que se muestran los datos obtenidos para cada consultas en cuanto a la precisión, recall y F-measure o F_1 score (los resultados se han redondeado a dos decimales):

Identificador consulta	Neo4j			RSHP		
	Precisión	Recall	F1	Precisión	Recall	F1
q1	1	0,07	0,13	1	0,03	0,06
q2	1	0,42	0,59	0,25	0,1	0,14
q3	0,8	0,12	0,21	1	0,15	0,26
q4	1	0,13	0,23	0	0	0
q5	1	0,83	0,91	0	0	0
q6	1	0,01	0,02	1	0,01	0,02
q7	1	0,01	0,02	0,9	0,01	0,02
q8	0,75	0,01	0,02	1	0,01	0,02
q9	1	0,01	0,02	0	0	0
q10	1	0,05	0,1	1	0,05	0,1
q11	1	0,02	0,04	1	0,02	0,04
q12	1	0,12	0,21	1	0,07	0,13
q13	1	0,15	0,26	1	0,15	0,26
q14	0,9	0,11	0,2	1	0,12	0,21
q15	1	0,13	0,23	0	0	0
q16	1	0,83	0,91	0,4	0,33	0,36
q17	1	0,01	0,02	1	0,01	0,02
q18	1	0,06	0,11	1	0,06	0,11
q19	0,8	0,01	0,02	1	0,01	0,02
q20	1	0,06	0,11	1	0,06	0,11
q21	1	0,09	0,17	1	0,09	0,17
q22	0,9	0,12	0,21	0	0	0
q23	1	0,01	0,02	1	0,01	0,02
q24	1	0,06	0,11	1	0,06	0,11
q25	1	0,15	0,26	1	0,15	0,26
q26	0,8	0,04	0,08	1	0,03	0,06
q27	1	0,01	0,02	1	0,01	0,02
q28	1	0,03	0,06	1	0,03	0,06
q29	1	0,05	0,1	1	0,05	0,1
q30	1	0,13	0,23	0,8	0,1	0,18
Media	**0,97**	**0,13**	0,23	**0,78**	**0,06**	0,11

Tabla 51: Resumen resultados obtenidos (Precisión, Recall y F-measure)

6.3 ANÁLISIS GLOBAL DE LOS RESULTADOS OBTENIDOS

Después de visualizar de manera rápida las tablas del apartado anterior, se intuye que los resultados que arroja Neo4j para el estudio realizado son mejores que los obtenidos por RSHP. En este apartado se van a analizar dichos resultados con el apoyo de distintos gráficos que ayudarán a leerlos y comprenderlos mejor.

En primer lugar, se van a comentar algunos aspectos de KM que han influido en que los resultados de RSHP no sean tan buenos como los de Neo4j. En las consultas 4, 9, 15 y 22, se puede observar que RSHP no devuelve ningún resultado para las consultas. Para estas cuatro consultas, esto es debido a que al indexar los requisitos en KnowledgeMANAGER, algunos términos se han indexado dentro de relaciones, de tal forma, que si se realiza una consulta por alguno de esos términos, el sistema no los reconoce dentro de los artefactos (requisitos), por lo que no se devuelve ningún resultado.

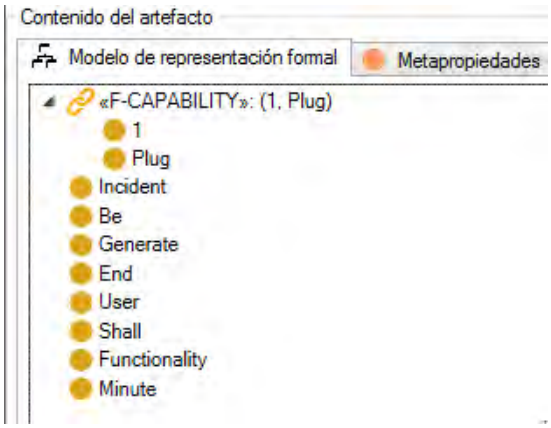


Figura 123: Representación formal artefacto

Como se puede comprobar en la *Figura 123*, uno de esos términos que KM ha incluido en relaciones, es el término “plug”. Si se realiza una consulta que sólo contenga ese término, KM no recuperará ningún resultado.

Otro aspecto reseñable que ha contribuido a que los resultados obtenidos en RSHP no sean los óptimos, tiene que ver con el modelo de representación formal que se realiza en KM de cada uno de los artefactos. Cuando se indexan los datos, KM crea el modelo de representación formal de cada artefacto generado (como se veía en la *Figura 117*) y descarta aquellos términos que no tienen un valor semántico destacable. Esto provoca que si se ejecuta una consulta que contiene un término que ha sido descartado de los artefactos, no se devuelvan ninguno de los requisitos que sí contienen ese término (recordar *Figura 119*).

Por otro lado, para algunas consultas como la 2, 5 y 16, entre otras, RSHP devuelve algunos resultados que no son relevantes o incluso, no incluye ningún resultado relevante entre los 20 primeros. Esto es debido a que, en KnowledgeMANAGER, existen clústeres que agrupan una serie de términos con algún tipo de relación. Por ejemplo, cuando se ejecutaron las consultas, se tenía un clúster denominado “communication” en el que se incluían algunos términos como “alert”, “say”, “warn” y “communicate”, como se puede comprobar en la siguiente imagen.

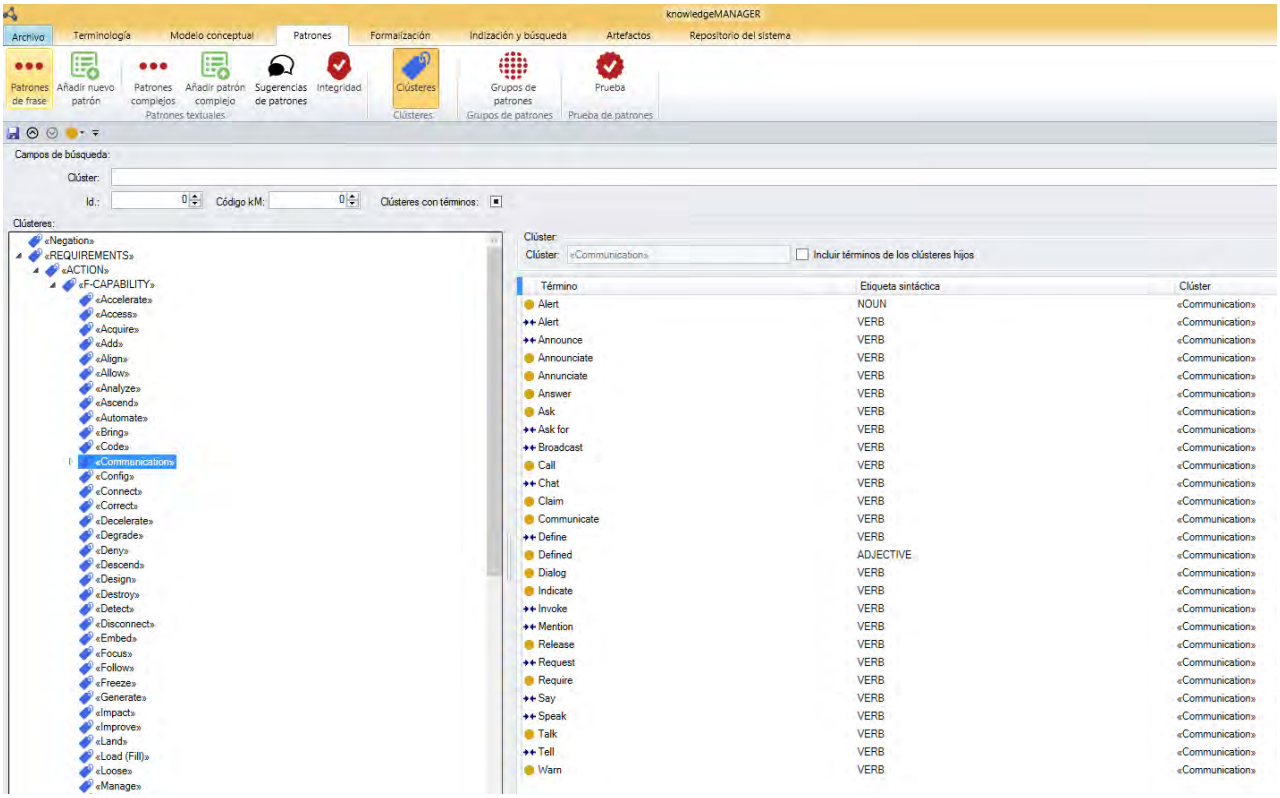


Figura 124: Clúster "Communication"

En esta situación, cuando se ejecuta una consulta que incluye alguno de los términos del clúster, en los resultados aparecerán los artefactos que incluyen el término de la consulta, pero también se devuelven aquellos artefactos que incluyen los otros términos comunes del clúster. Como para el presente estudio se pretenden recuperar requisitos que contengan alguno de los términos de la consulta, todos aquellos artefactos recuperados que incluyen otros términos del clúster, son considerados como no relevantes para la consulta. Por este motivo, algunas de las consultas no devuelven los requisitos esperados.

Por su parte, Neo4j ofrece unos resultados destacables en lo que se refiere a la recuperación de los requisitos relevantes para cada consulta. Se puede comprobar como para algunas consultas, la precisión baja levemente en Neo4j. Esta bajada se debe a la acumulación de términos en nodos, que se veía en el capítulo de *Diseño de la solución final*. Con los gráficos que se incluyen a continuación, se comparan los resultados arrojados por ambos sistemas, incluyendo las medidas de precisión y recall.

Seguidamente, en la *Figura 125* se muestra un gráfico en el que se representa cómo varía la precisión de cada sistema en las 30 consultas que abarca el experimento:

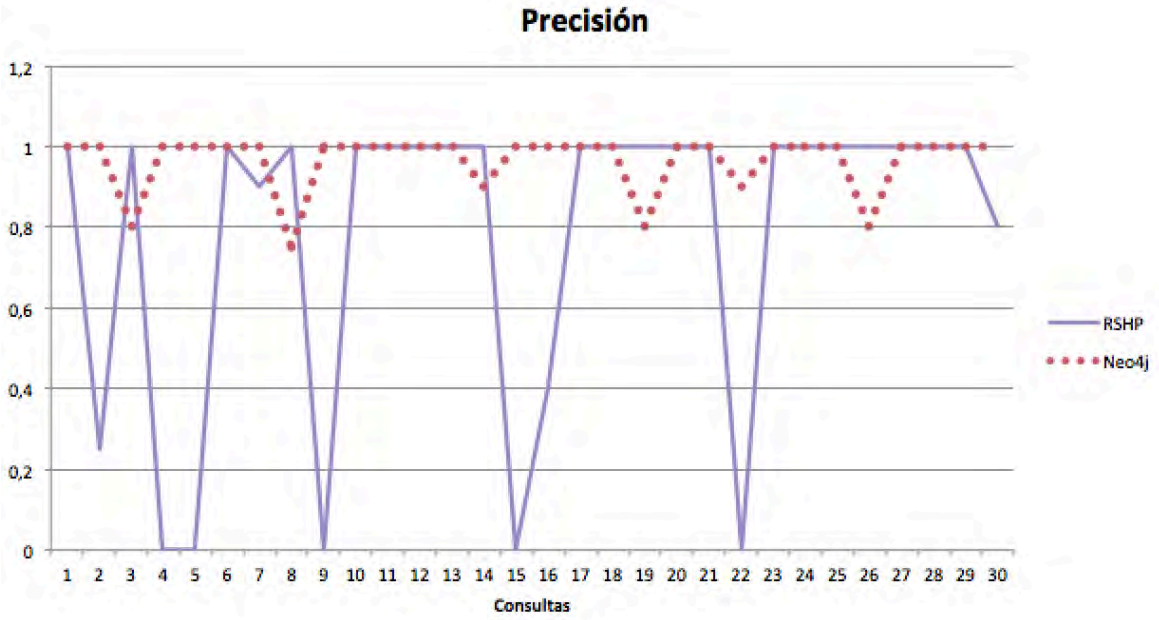


Figura 125: Precisión 30 consultas en RSHP y Neo4j

A partir de la *Figura 125*, se comprueba como Neo4j mantiene una precisión muy constante manteniéndose siempre entre el 75% y el 100%. Esto quiere decir que Neo4j introduce poco ruido en sus resultados. Como se ha comentado, esas pequeñas bajadas en la precisión de algunas consultas en Neo4j, es provocada por la acumulación de términos en los nodos. Sin embargo, la precisión de RSHP es más variable ya que, en 5 de las consultas ejecutadas, no se ha obtenido ningún requisito relevante para la consulta. Es decir, RSHP introduce bastante más ruido que Neo4j en sus resultados, lo que provoca que la precisión baje y se obtenga un gráfico cambiante. Esta bajada en la precisión de las consultas en RSHP es provocada por los distintos aspectos que se han comentado anteriormente.

En el siguiente gráfico se muestra cómo varía la recall en cada uno de los sistemas:

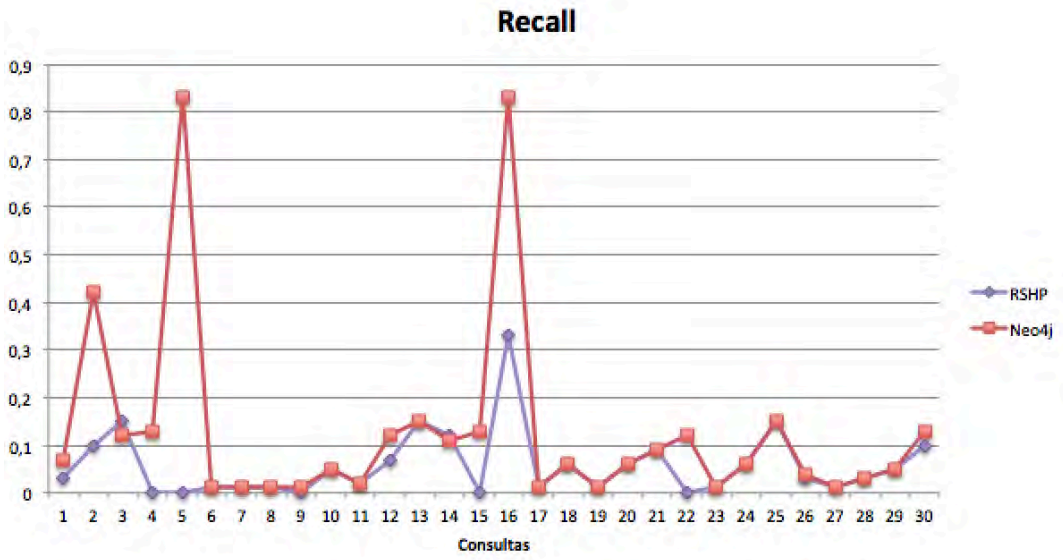


Figura 126: Recall 30 consultas en RSHP y Neo4j

Como se puede observar en el gráfico de la *Figura 126*, la recall de las consultas en Neo4j es, en líneas generales, mejor que en RSHP. Como se puede apreciar, en algunas de las consultas la recall es la misma para ambos sistemas, pero Neo4j destaca muy por encima de RSHP en otras como las consultas número 2, 5 y 16. Cabe destacar que en la mayoría de las consultas se han obtenido valores bajos, tanto para Neo4j como para RSHP. Esto es debido a que, por regla general, el número de requisitos relevantes para cada una de las consultas, es muy elevado. Por lo que, por lo expuesto, se puede afirmar que en los resultados de RSHP existe mayor silencio que en los arrojados por Neo4j. Cuanto menor es el silencio mejor, ya que significa que se han recuperado muchos de los resultados relevantes. En este sentido, Neo4j ofrece mejores datos.

A continuación se van a mostrar los mismos datos de precisión y recall pero en distintos gráficos, teniendo en cuenta el número de términos de las consultas. De esta manera, se pretende conocer si la precisión y la recall mejoran o empeoran según se aumenta el número de términos de la consulta. En primer lugar, se muestran los tres gráficos correspondientes a la precisión:

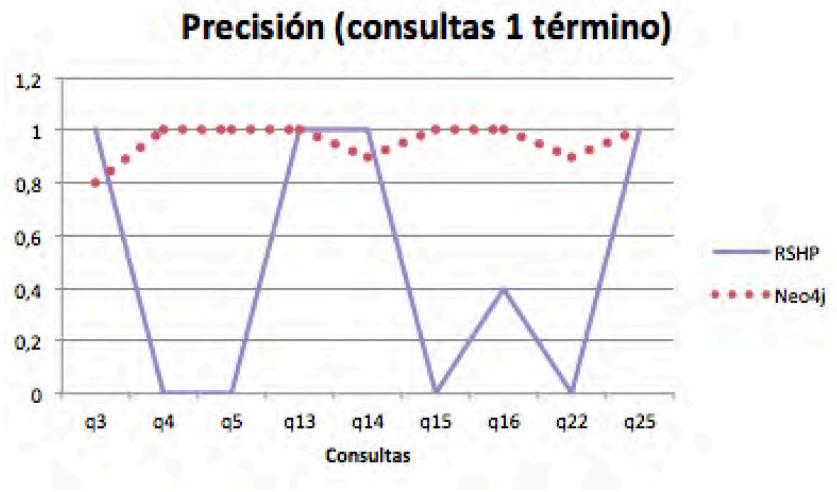


Figura 127: Precisión consultas 1 término en RSHP y Neo4j

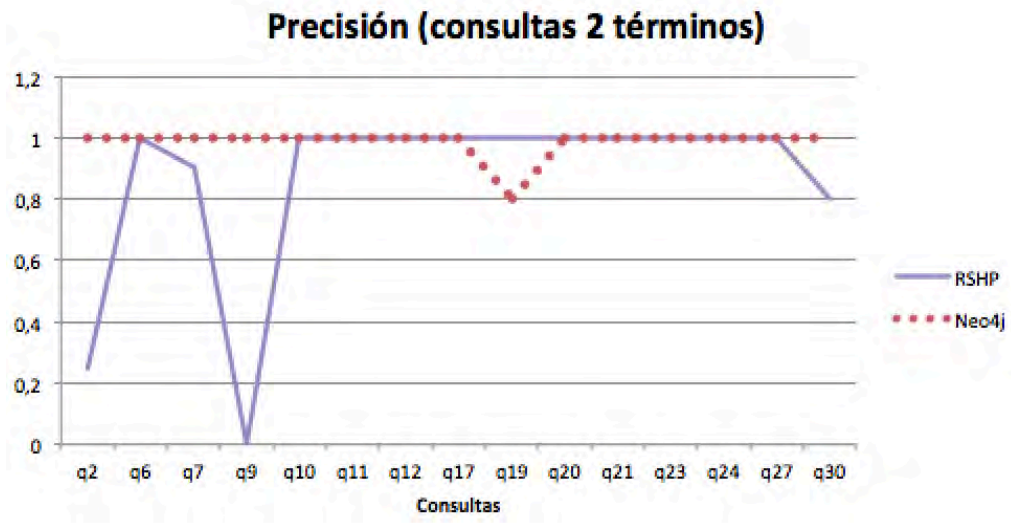


Figura 128: Precisión consultas 2 términos en RSHP y Neo4j

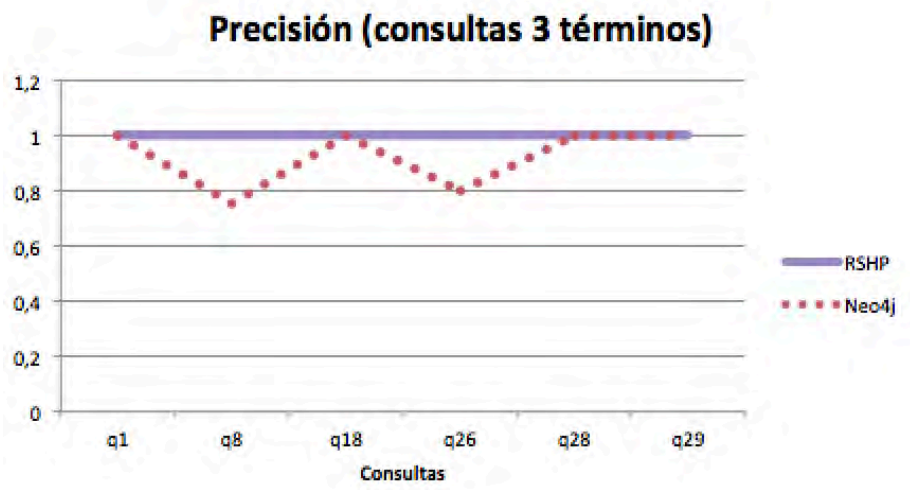


Figura 129: Precisión consultas 3 términos en RSHP y Neo4j

A partir de las *Figuras 127, 128* y *129*, se comprueba como a medida que se aumenta el número de términos de las consultas, la precisión que se obtiene en RSHP se ve mejorada de forma notoria. Para las consultas de un término, se obtiene un gráfico cambiante, sin mostrar un ápice de constancia. Pero para las consultas de 2 términos, a pesar de que la precisión baje en algunas consultas, se mantiene una constancia importante de 100% de precisión en 10 consultas. Por su parte, todas las consultas de 3 términos arrojan una precisión del 100%. Si bien es cierto que se han ejecutado un mayor número de consultas de 2 términos que de 1 y 3, sí que se puede extraer que existe una mejoría importante en la precisión de RSHP, a medida que se aumentan el número de términos de las consultas. En cuanto a la precisión de las consultas en Neo4j, muestra un comportamiento similar en los tres gráficos, manteniéndose siempre bastante constante.

Como se ha comentado anteriormente, a continuación se incluyen los gráficos correspondientes a la recall, realizando la misma división que para la precisión:

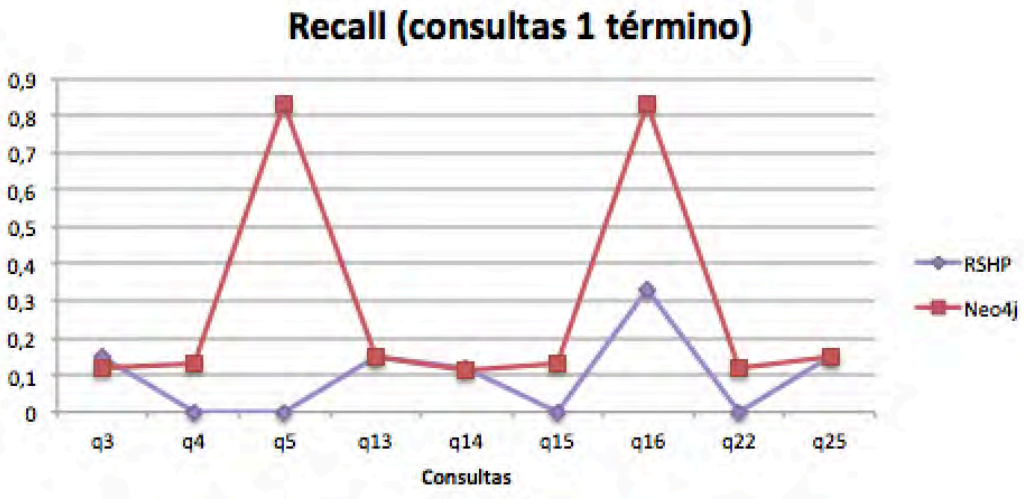


Figura 130: Recall consultas 1 término en RSHP y Neo4j

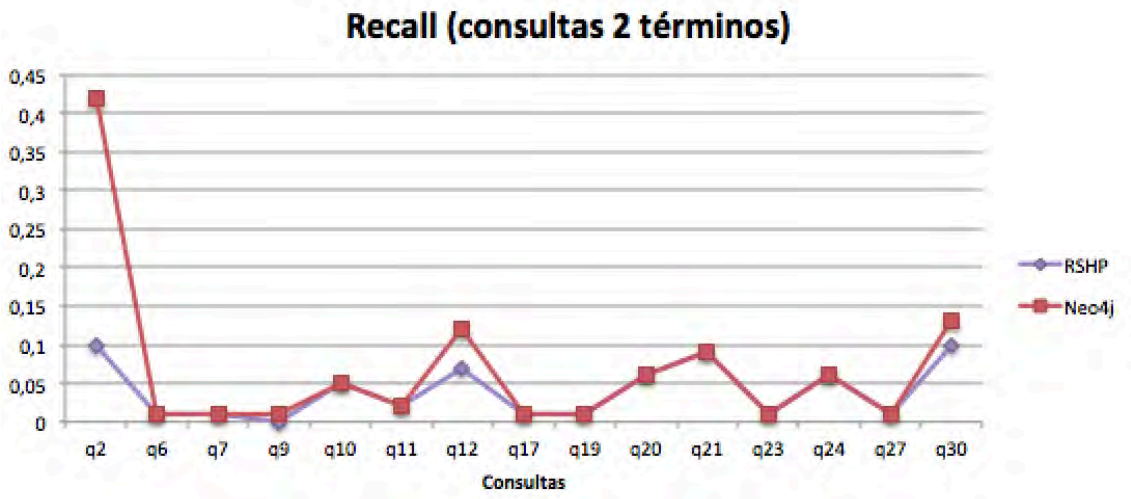


Figura 131: Recall consultas 2 términos en RSHP y Neo4j

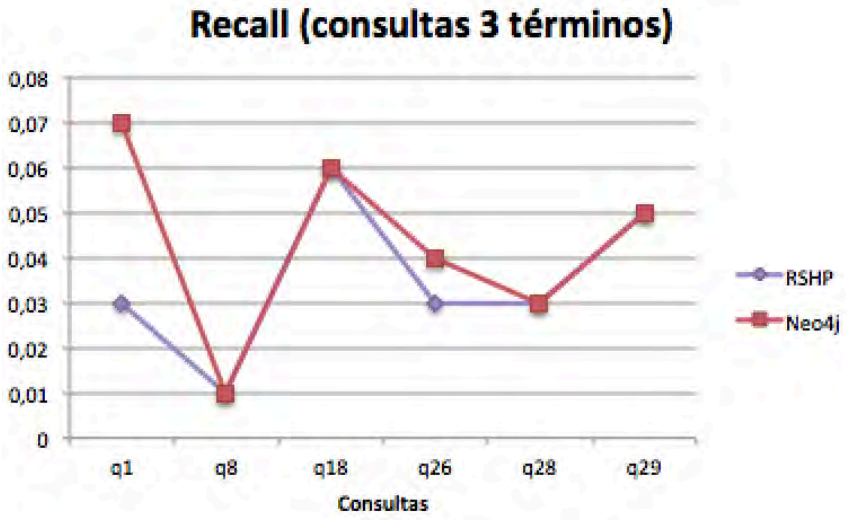


Figura 132: Recall consultas 3 términos en RSHP y Neo4j

A partir de las 3 figuras anteriores, se puede observar como las mayores diferencias de recall entre RSHP y Neo4j, vienen dadas por consultas de un sólo término. Además, en esas consultas la recall no baja de 0,1 (excepto alguna consulta en RSHP). Para las consultas de 2 términos, los resultados se mantienen mucho más equilibrados entre ambos sistemas y la gran mayoría de los valores se encuentran por debajo de 0,1 de recall. Las consultas de 3 términos, al igual que las de dos, también ofrecen un mayor equilibrio entre los dos sistemas. Además, cabe destacar que para este tipo de consultas los valores de la recall bajan de forma drástica siendo 0,07 el valor más alto.

Por lo que, según lo comentado, se puede abstraer que a mayor número de términos en las consultas, más bajos e igualados son los valores de la recall, para RSHP y Neo4j. Esto es previsible, ya que cuantos más términos se incluyan en la consulta, seguramente, mayor será el número de resultados relevantes y, por tanto, se obtendrán valores más bajos e igualados (se recuerda que para este estudio se ha realizado un cut-off a los 20 primeros resultados).

Con los tres siguientes gráficos, se pretende averiguar si el número de términos de las consultas afecta, de alguna forma, a la recuperación de los resultados esperados en ambos sistemas.

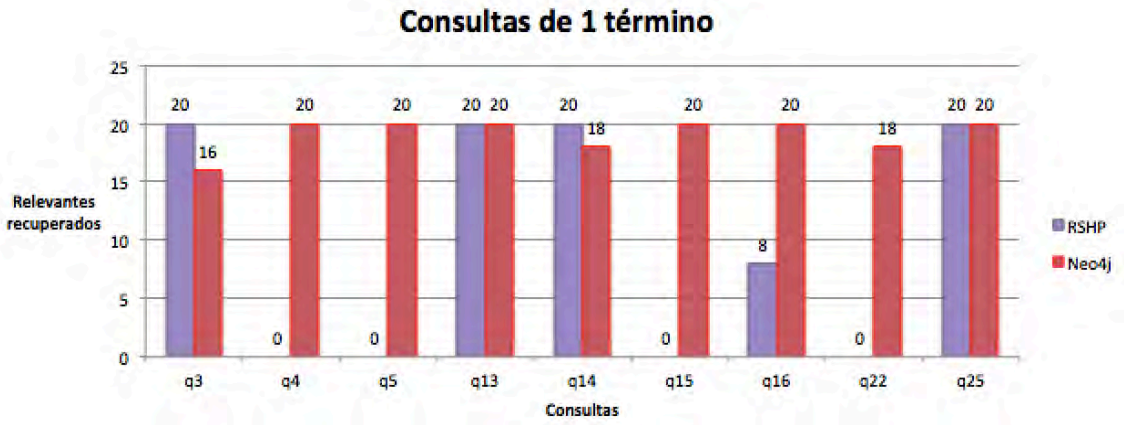


Figura 133: Requisitos relevantes recuperados (consultas 1 término)

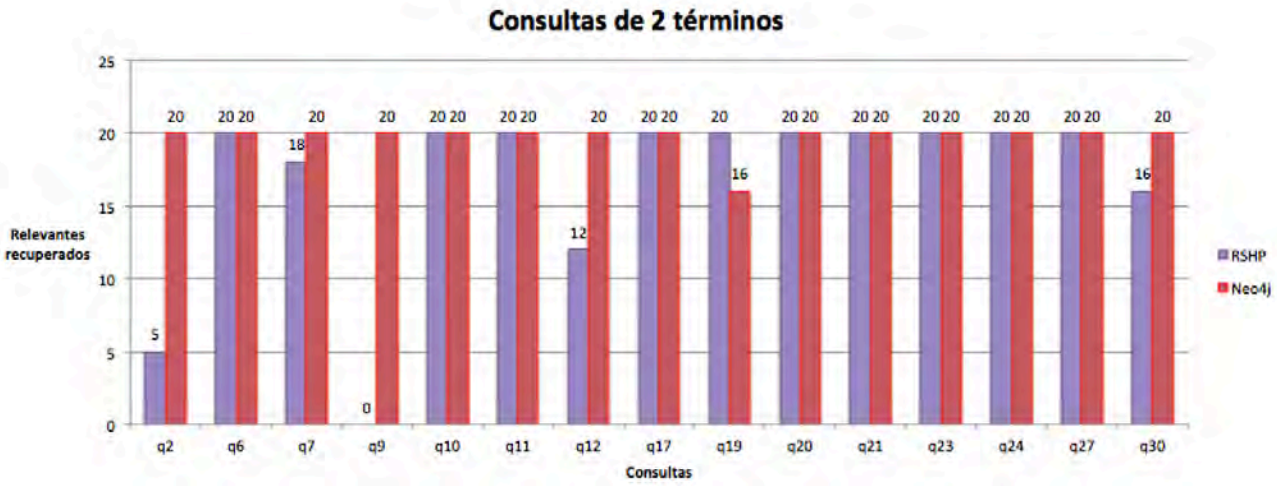


Figura 134: Requisitos relevantes recuperados (consultas 2 términos)

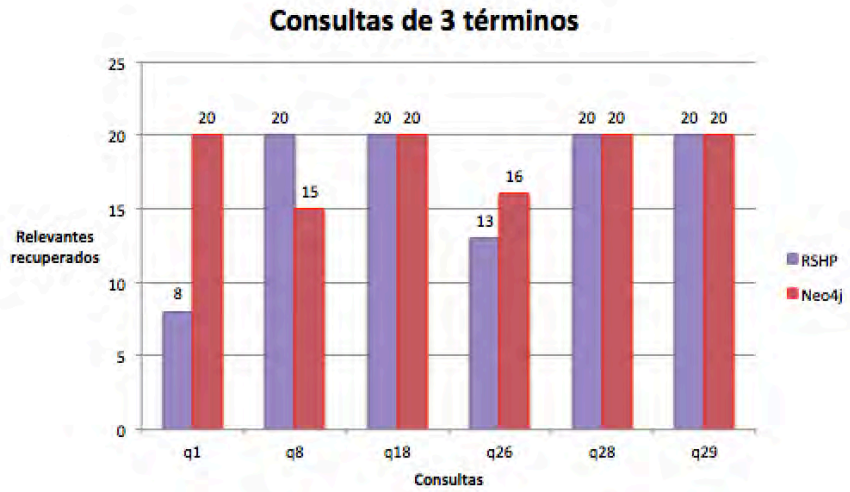


Figura 135: Requisitos relevantes recuperados (consultas 3 términos)

A partir de las *Figuras 133, 134* y *135*, se puede comprobar como existe una notable mejoría, en cuanto a los requisitos relevantes recuperados en RSHP, en las consultas con 2 y 3 términos respecto de las consultas que sólo implican un término. En el gráfico correspondiente a las consultas de un término (*Figura 133*), ni la mitad de las consultas RSHP recuperan 20 relevantes dentro de los 20 primeros resultados. Además, 4 de las 9 consultas de un término no devuelven ningún requisito relevante entre los 20 primeros resultados.

Sin embargo, en las consultas de 2 términos (*Figura 134*), los resultados son mucho mejores ya que de 15 consultas ejecutadas, en 10 de ellas los 20 primeros resultados son todos relevantes. Adicionalmente, sólo aparece una consulta que no devuelva ningún resultado relevante. Esto supone una importante mejora por parte de RSHP respecto de las consulta de un único término.

Por su parte, las consultas de 3 términos (*Figura 135*), parecen seguir la dinámica de las consultas de 2 elementos ya que de 6 consultas, en 4 de ellas los 20 primeros resultados son relevantes. Además, todas ellas devuelven requisitos relevantes entre los 20 primeros.

Por lo que, según lo expuesto, se llega a la conclusión de que, en RSHP se recuperan mejor los requisitos relevantes para consultas que contengan, al menos, 2 términos. Para consultas de un solo término la recuperación en RSHP es bastante pobre. Por su parte, Neo4j se mantiene constante en el número de requisitos relevantes recuperados como se puede observar en las tres figuras anteriores.

Por todo lo comentado hasta ahora, se concluye que **para el presente estudio, Neo4j arroja mejores resultados que RSHP**, si bien es cierto que para consultas de 3 términos, los resultados mejoran de forma notable en RSHP, siendo mucho más similares a los obtenidos en Neo4j. Como se ha podido comprobar, Neo4j (utilizando el diseño expuesto en la sección *Diseño de la solución final*) responde muy bien y de forma constante, en cuanto a la recuperación de elementos en base a términos concretos, manteniendo una precisión muy alta en todas las consultas. Por su parte, RSHP no ofrece unos malos resultados, pero sí que existe un mayor margen para la mejora.

Por tanto, queda evidenciado que para consultas sobre términos concretos, Neo4j se amolda mejor que RSHP, el cual adquiere mayor potencial para consultas inteligentes. Es decir, RSHP posee un mayor potencial para consultas algo más generales, en las que los resultados relevantes no sean sólo aquellos que contienen algún término de la consulta. La aplicación del conocimiento adquirido, en RSHP, permite ofrecer unos resultados más interesantes. En este caso, RSHP aplica el conocimiento obtenido (por ejemplo, a través de clústeres), lo cual provoca que recuperen resultados no relevantes para este estudio.

Aunque Neo4j obtiene mejores resultados, la flexibilidad de RSHP para realizar consultas en lenguaje natural permite asumir esa variación en la precisión ya que está más cerca a lo que el usuario puede formular.

6.4 RECALCULANDO REQUISITOS RELEVANTES A LAS CONSULTAS

En este apartado se pretende comprobar si los resultados que arroja RSHP mejoran, si se consideran relevantes todos aquellos requisitos que contienen, en vez del término de la consulta, algún término perteneciente a algún clúster.

Como se veía en el apartado anterior, KM dispone de numerosos clúster que agrupan términos que guardan cierta similitud. Si el término de la consulta pertenece a algún clúster, RSHP recupera todos los requisitos que contienen el término de consulta, pero también todos aquellos requisitos que contienen algún término del clúster común.

Como para el experimento realizado sólo eran considerados como relevantes aquellos requisitos que contuviesen el término o términos de consulta, algunas consultas en RSHP bajaban su precisión de manera importante. Por lo que en este apartado, se van a considerar requisitos relevantes tanto a los que contienen términos de la consulta, como aquellos que pertenezcan a algún clúster común con los términos de la consulta. De esta forma, se va a comprobar cómo varían las medidas de precisión y recall.

En primer lugar, cabe destacar que las consultas que mejoran su precisión con esta modificación son las siguientes: consulta 2, 5, 16 y 30. A continuación se muestra dos gráficos, para comparar los requisitos relevantes recuperados anteriormente y ahora realizando esta modificación para RSHP:

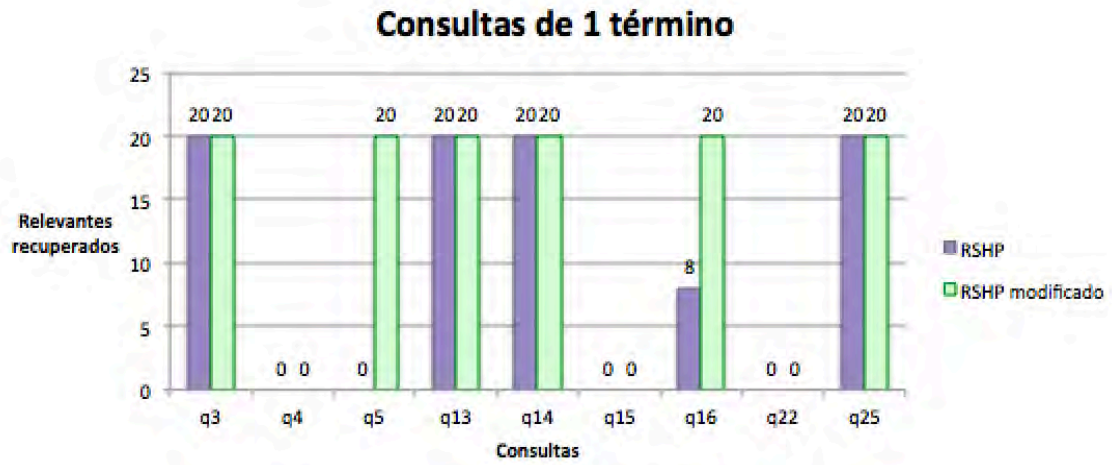


Figura 136: Requisitos relevantes recuperados consultas de 1 término (RSHP vs RSHP modificado)

A partir de la *Figura 136*, se comprueba como en las consultas 5 y 16 de un único término, se pasa de obtener 0 y 8 resultados relevantes, entre los 20 primeros resultados, a tener 20 y 20, respectivamente.

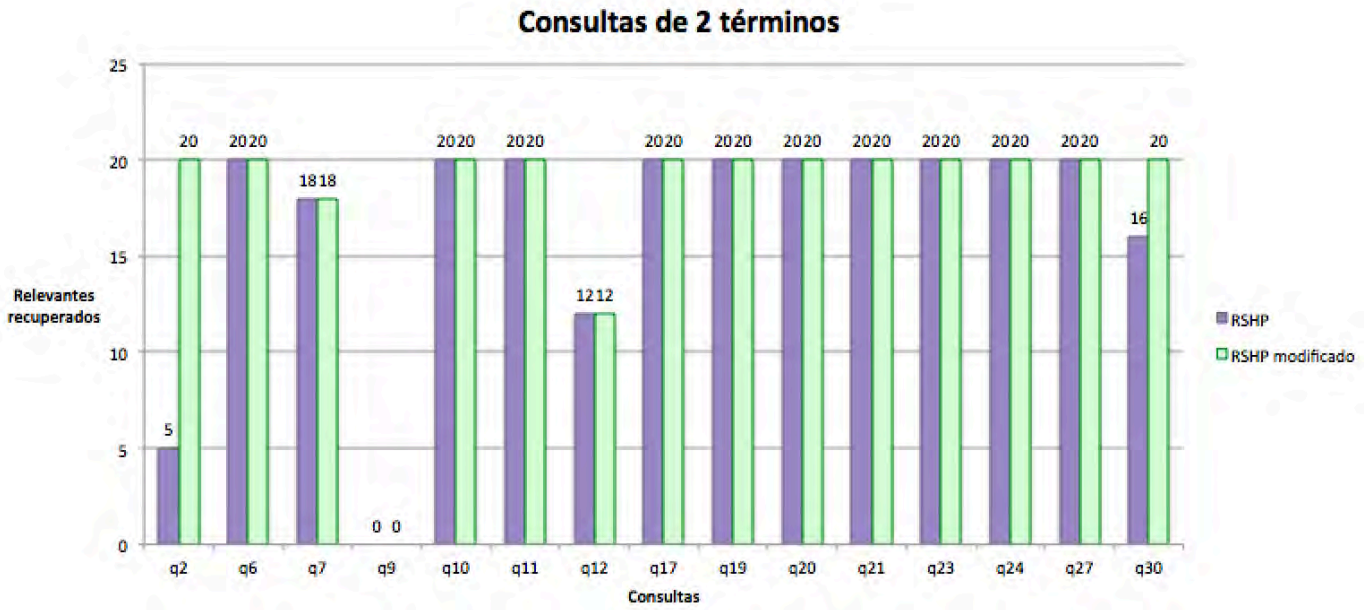


Figura 137: Requisitos relevantes recuperados consultas de 2 términos (RSHP vs RSHP modificado)

Como se puede comprobar en la *Figura 137*, las consultas 2 y 30 de dos términos, pasan de recuperar 5 y 16 requisitos relevantes entre los 20 primeros resultados, a recuperar 20 y 20, respectivamente. Por su parte, el gráfico correspondiente a consultas de 3 términos no sufre variaciones.

Esto provoca que las medidas de precisión y recall mejoren para dichas consultas en RSHP, como se puede comprobar en los siguientes gráficos:

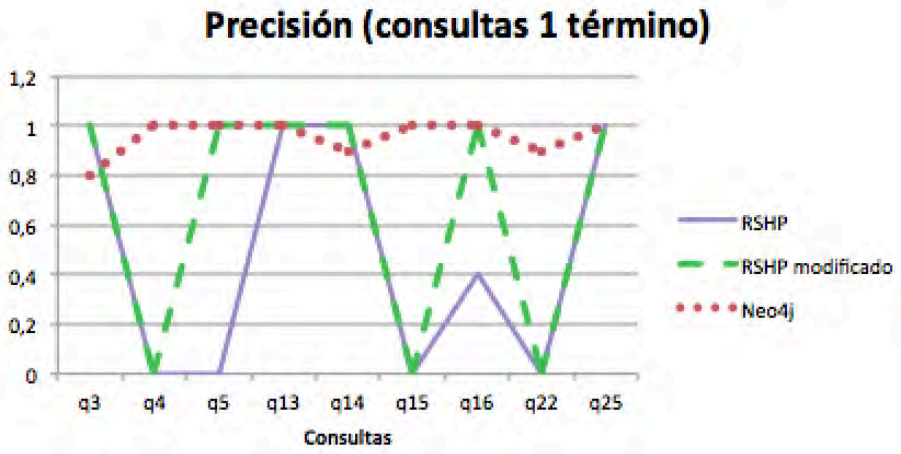


Figura 138: Precisión consultas 1 término (RSHP, RSHP modificado y Neo4j)

Como se puede apreciar en la *Figura 138*, para la consulta 5 se pasa de una precisión del 0% a una del 100% y para la consulta 16, de una precisión del 40%, se pasa a una del 100%. La línea correspondiente a RSHP modificado coincide más con la de Neo4j que la de RSHP, por lo que los resultados para RSHP mejorarían levemente.

Lo mismo sucede para las consultas de 2 términos, como se puede ver en el siguiente gráfico:

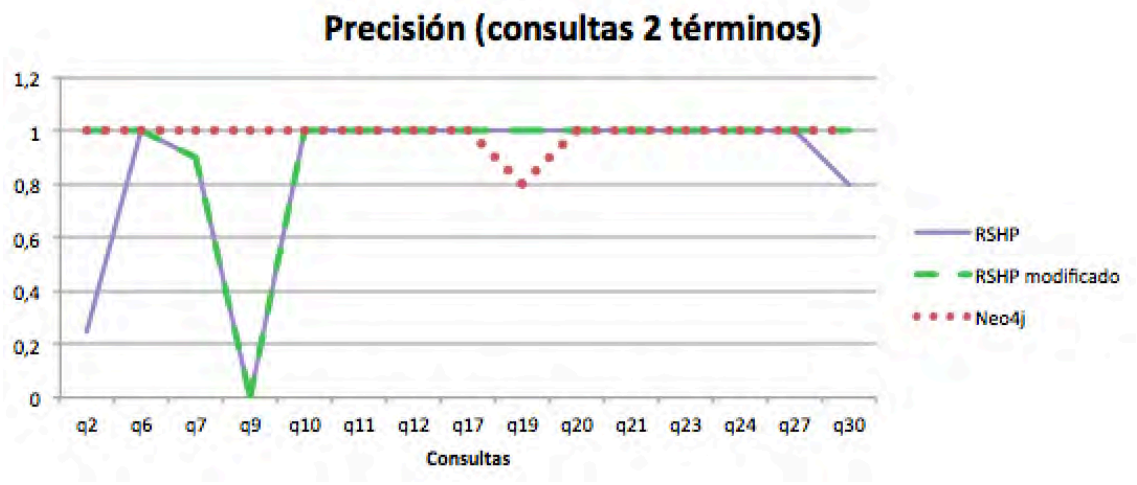


Figura 139: Precisión consultas 2 términos (RSHP, RSHP modificado y Neo4j)

Como se puede comprobar en la *Figura 139*, la consulta 2 pasa de una precisión del 25%, a una del 100% y la consulta 30, pasa de un 80% a un 100%.

A continuación se muestra cómo varía la recall para los nuevos datos:

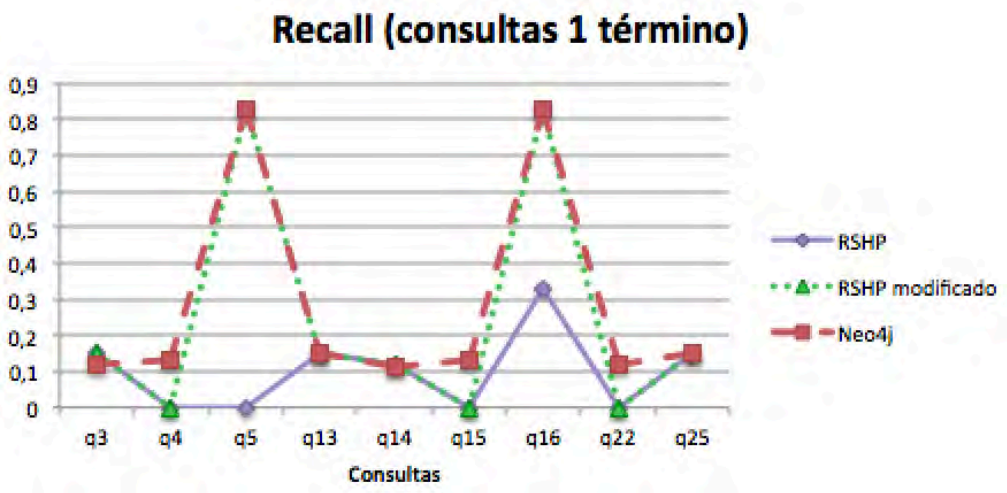


Figura 140: Recall consultas 1 término (RSHP, RSHP modificado y Neo4j)

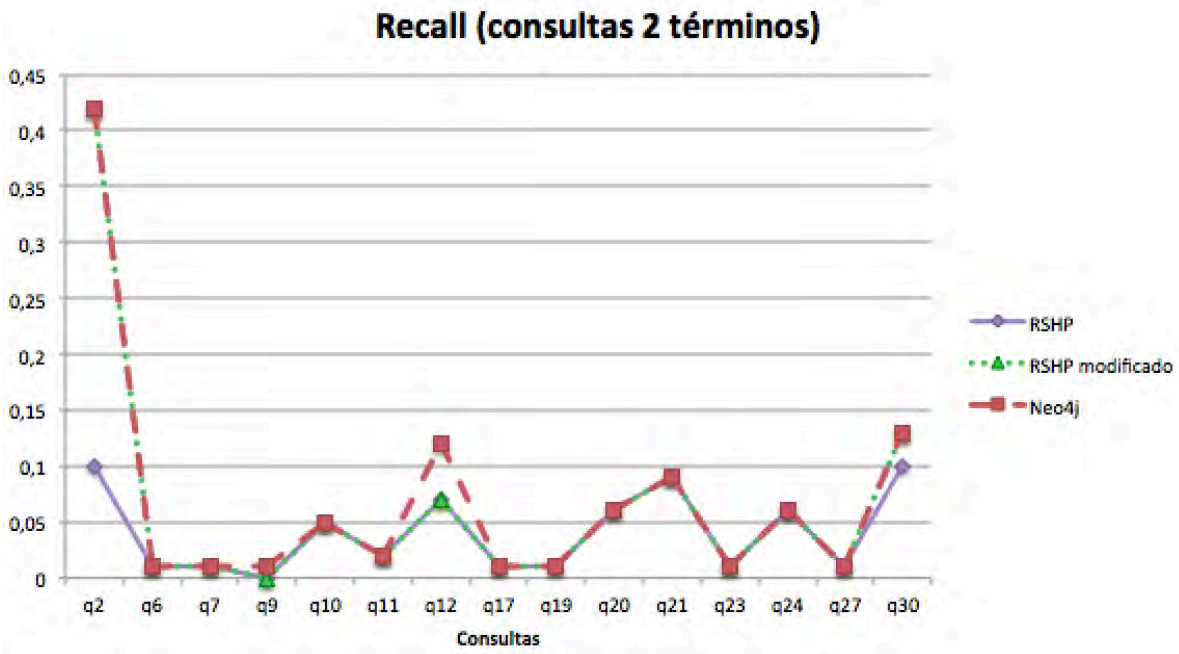


Figura 141: Recall consultas 2 términos (RSHP, RSHP modificado y Neo4j)

A partir de la *Figura 140*, se comprueba como para las consultas 5 y 16, la recall mejora de forma notoria, ya que se pasa de una recall del 0% y 33%, a una del 83% y 83%, respectivamente. Por su parte, las consultas 2 y 30 de la *Figura 141*, pasan de tener una recall del 10% y 10%, a una del 42% y 13%, respectivamente.

Con todo lo expuesto en este apartado, se demuestra que RSHP mejora las medidas de precisión y recall cuando se consideran relevantes requisitos que contienen otros términos relacionados con los de la consulta. De esta forma, RSHP se acerca más a los resultados que arroja Neo4j, aunque todavía para algunas consultas, no se obtienen resultados relevantes debido a los términos que se indexan en relaciones y a los términos que se eliminan de la representación formal de los artefactos en KM, como se veía en el apartado anterior de *Análisis de los resultados obtenidos*.

7. GESTIÓN DEL PROYECTO

En este capítulo se van a incluir dos apartados, en el primero de ellos se detallará la planificación realizada para el presente trabajo, mientras que en el segundo se incluirá un presupuesto detallado del mismo.

7.1 PLANIFICACIÓN DEL PROYECTO

En este apartado se incluye la planificación realizada para el presente Trabajo de Fin de Grado, incluyendo las fases y tareas identificadas en el mismo. A continuación, se indican cada una de esas fases que se han ido siguiendo, detallando, a su vez, las principales tareas que componen cada una de ellas:

- **Formación en Neo4j**
  - ✓ Búsqueda de información acerca de Neo4j
  - ✓ Realización curso online en la web oficial de Neo4j

- **Presentación sobre Neo4j**
  - ✓ Realización PowerPoint sobre Neo4j y Cypher
  - ✓ Edición de un vídeo presentando el PowerPoint
  - ✓ Publicación del vídeo-presentación en Youtube[38]

- **Planteamiento**
  - ✓ Definición de objetivos

- **Estado del arte**
  - ✓ Sistemas de almacenamiento basados en grafos
  - ✓ Benchmarks existentes entre distintos sistemas
  - ✓ Lenguajes de recuperación de información
  - ✓ Modelos de recuperación de información

- **Análisis**
  - ✓ Evaluación de distintas opciones de almacenamiento
  - ✓ Evaluación de la generación y ejecución de consultas

- **Diseño**
  - ✓ Arquitectura de componentes
  - ✓ Evaluación distintas alternativas de diseño
  - ✓ Definición diseño final del grafo en Neo4j

- **Implementación**
  - ✓ Creación del grafo en Neo4j
  - ✓ Generación y ejecución automática de consultas en Neo4j

- **Pruebas**
  - ✓ Pruebas de la implementación de cada diseño

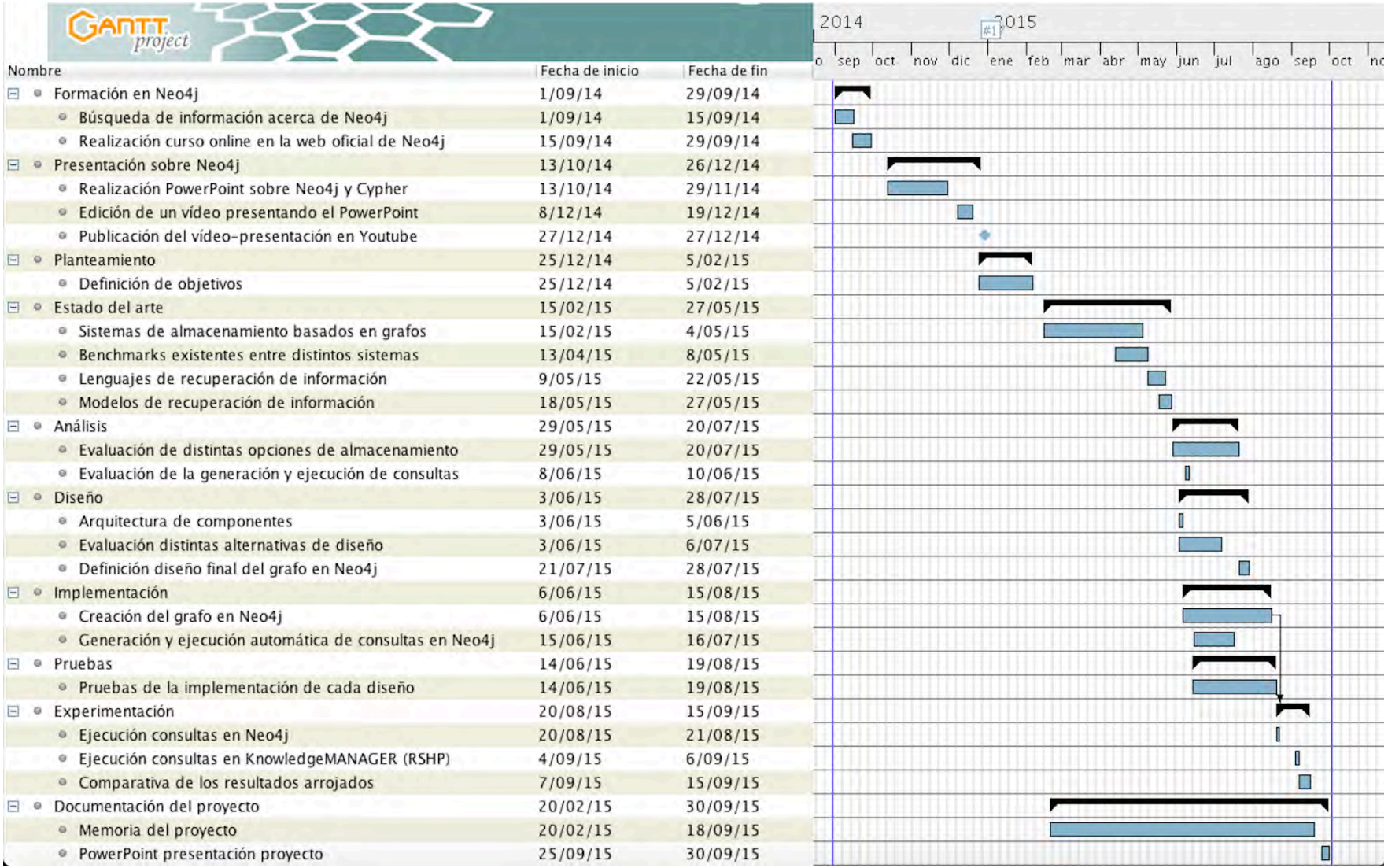
[38] La presentación se puede ver en la siguiente dirección: https://www.youtube.com/watch?v=EstGVbNe8eE

- **Experimentación**
  - ✓ Ejecución consultas en Neo4j
  - ✓ Ejecución consultas en KnowledgeMANAGER (RSHP)
  - ✓ Comparativa de los resultados arrojados

- **Documentación del proyecto**
  - ✓ Memoria del proyecto
  - ✓ PowerPoint presentación proyecto

En cuanto a las tareas críticas, cabe destacar una tarea que es completamente necesaria para poder continuar con el estudio, esa tarea es *Creación del grafo en Neo4j*, por lo que se convierte en una tarea crítica. Asimismo, las tareas *Ejecución consultas en Neo4j* y *Ejecución consultas en KnowledgeMANAGER (RSHP),* también son tareas críticas ya que su realización es fundamental para efectuar la comparativa entre ambos sistemas. El resto de tareas son importantes, pero no son consideradas como críticas.

Después de indicar las principales fases y tareas identificadas para el desarrollo de este trabajo y de identificar las tareas críticas, se incluye la secuenciación y temporización de cada una de ellas mediante un diagrama de Gantt, el cual se muestra a continuación:



Nombre	Fecha de inicio	Fecha de fin
Formación en Neo4j	1/09/14	29/09/14
Búsqueda de información acerca de Neo4j	1/09/14	15/09/14
Realización curso online en la web oficial de Neo4j	15/09/14	29/09/14
Presentación sobre Neo4j	13/10/14	26/12/14
Realización PowerPoint sobre Neo4j y Cypher	13/10/14	29/11/14
Edición de un vídeo presentando el PowerPoint	8/12/14	19/12/14
Publicación del vídeo-presentación en Youtube	27/12/14	27/12/14
Planteamiento	25/12/14	5/02/15
Definición de objetivos	25/12/14	5/02/15
Estado del arte	15/02/15	27/05/15
Sistemas de almacenamiento basados en grafos	15/02/15	4/05/15
Benchmarks existentes entre distintos sistemas	13/04/15	8/05/15
Lenguajes de recuperación de información	9/05/15	22/05/15
Modelos de recuperación de información	18/05/15	27/05/15
Análisis	29/05/15	20/07/15
Evaluación de distintas opciones de almacenamiento	29/05/15	20/07/15
Evaluación de la generación y ejecución de consultas	8/06/15	10/06/15
Diseño	3/06/15	28/07/15
Arquitectura de componentes	3/06/15	5/06/15
Evaluación distintas alternativas de diseño	3/06/15	6/07/15
Definición diseño final del grafo en Neo4j	21/07/15	28/07/15
Implementación	6/06/15	15/08/15
Creación del grafo en Neo4j	6/06/15	15/08/15
Generación y ejecución automática de consultas en Neo4j	15/06/15	16/07/15
Pruebas	14/06/15	19/08/15
Pruebas de la implementación de cada diseño	14/06/15	19/08/15
Experimentación	20/08/15	15/09/15
Ejecución consultas en Neo4j	20/08/15	21/08/15
Ejecución consultas en KnowledgeMANAGER (RSHP)	4/09/15	6/09/15
Comparativa de los resultados arrojados	7/09/15	15/09/15
Documentación del proyecto	20/02/15	30/09/15
Memoria del proyecto	20/02/15	18/09/15
PowerPoint presentación proyecto	25/09/15	30/09/15

Figura 142: Diagrama de Gantt

Como se puede comprobar en la *Figura 142*, se incluye el diagrama de Gantt de la planificación del presente Trabajo de Fin de Grado, el cual se ha realizado mediante la herramienta GanttProject[39], en su versión 2.7.1 para Mac OS X. Como se puede observar, se incluyen las fases y tareas que se indicaron anteriormente, especificando las fechas de inicio y fin de cada una de ellas. Teniendo en cuenta la fecha de inicio de la primera fase y la fecha de fin de la última, se deduce que la fecha de inicio de este estudio es 01/09/14, mientras que la fecha de fin es 30/09/15.

Además, en la parte derecha de la *Figura 142*, se puede ver el diagrama, en el que cada barra azul representa la duración de cada tarea y cada llave negra representa la duración completa de cada fase. Por su parte, el rombo azul indica que la tarea *Publicación del vídeo-presentación en Youtube* es un hito. Las líneas verticales marcan el comienzo y fin del proyecto.

Como se puede apreciar, hasta la fase de análisis, según finalizaba una fase comenzaba otra. Pero las fases *Análisis*, *Diseño*, *Implementación* y *Pruebas* se mantienen de forma simultánea durante cierto espacio de tiempo. Esto es debido a que se han ido realizando distintas versiones del grafo en Neo4j, realizando para cada una de ellas su correspondiente análisis, diseño, implementación y pruebas. Este modo de trabajar coincide, en gran medida, con los modelos de desarrollo software evolutivos, como por ejemplo, el modelo en espiral.

Por último, después de finalizar las pruebas de la implementación del diseño final, comienza la fase de experimentación. Cabe destacar que la redacción de esta memoria comenzó unos días después del inicio de la fase Estado del arte, alargándose en el tiempo hasta el 18 de Septiembre de 2015.

[39] http://www.ganttproject.biz/

7.2 PRESUPUESTO DEL PROYECTO

En este apartado se elabora el presupuesto del presente Trabajo de Fin de Grado. Para ello, se han tenido en cuenta los diferentes gastos acometidos durante el mismo (costes materiales). De la misma forma, se ha tenido en cuenta el tiempo dedicado a la realización del trabajo para calcular los costes de recursos humanos. Los costes que se van a desglosar en distintos apartados son los siguientes:

- Costes de recursos humanos
- Costes de material
- Costes indirectos

Cabe destacar que la unidad monetaria utilizada es el Euro (€) y que tanto en los cálculos de los costes comentados, como en los del coste total del proyecto, se tendrá en cuenta un redondeo con dos decimales. Asimismo, se aplicarán los impuestos que sean oportunos.

7.2.1 COSTES DE RECURSOS HUMANOS

Para calcular los costes de recursos humanos, primero hay que contabilizar las horas que se han dedicado a la realización del trabajo. A continuación, se muestra el tiempo dedicado a la realización de este trabajo en cada uno de los meses que abarcaba la planificación detallada en el capítulo anterior. En la siguiente tabla se indican el número de semanas que se ha trabajado en cada mes y el tiempo dedicado, de media, en cada una de esas semanas:

Mes	Semanas trabajadas	Horas por semana	Horas Totales
Septiembre	3	10	30
Octubre	2	20	40
Noviembre	2	20	40
Diciembre	2	20	40
Enero	1	5	5
Febrero	2	10	20
Marzo	3	10	30
Abril	3	20	60
Mayo	1	20	20
Junio	4	20	80
Julio	4	30	120
Agosto	4	20	80
Septiembre	2	20	40
TOTAL			**605**

Tabla 52: Tiempo dedicado

En la *Tabla 52* se puede observar que las horas totales dedicadas al proyecto han sido 605. De forma adicional, cabe destacar que se han dedicado 25 horas en una serie de reuniones con José María Álvarez Rodríguez (cotutor de este TFG) para consultar algunas dudas. No obstante, estas horas extra no se van a tener en cuenta para realizar los cálculos, ya que no han supuesto un añadido en el coste de total de recursos humanos.

Sabiendo ya las horas totales dedicadas, se procede a calcular los costes asociados a los recursos humanos. Para ello, sólo se va a tener en cuenta, como autor de este Trabajo de Fin de Grado, a Roberto Monsalve Toledo. A continuación, en la *Tabla 53*, se muestran los costes asociados al personal, teniendo en cuenta las horas totales dedicadas calculadas en la tabla anterior y asumiendo un coste de 15€/hora de un Ingeniero Junior:

Personal	Categoría	Coste persona (€/hora)	Horas dedicadas	Coste (€)
Roberto Monsalve Toledo	Ingeniero Junior	15,00	605	9.075,00

Tabla 53: Costes personal

Como se puede comprobar en la *Tabla 53*, los costes asociados al personal ascienden a 9.075,00€.

Seguidamente, se indican un par de cuestiones importantes a la hora de calcular los costes totales de este apartado. En primer lugar, hay que tener en cuenta las bases de cotización según la categoría profesional del personal implicado en el trabajo. A continuación, en la *Tabla 54*, se muestra un extracto de la tabla de bases de cotización de 2015 de la página web del Ministerio de Empleo y Seguridad Social[40]:

Bases de cotización contingencias comunes			
Grupo de cotización	**Categorías profesionales**	**Bases mínimas (€/mes)**	**Bases máximas (€/mes)**
1	Ingenieros y Licenciados. Personal de alta dirección no incluido en el artículo 1.3.c) del Estatuto de los Trabajadores	1.056,90	3.606,00
2	Ingenieros Técnicos, Peritos y Ayudantes titulados	876,60	3.606,00

Tabla 54: Bases de cotización 2015

Por otro lado, también hay que tener en cuenta los tipos de cotización. En la *Tabla 55* se muestran los tipos de cotización de 2015, distinguiendo entre la parte asumida por la empresa y la asumida por el trabajador. Esta información también ha sido extraída de la página web del Ministerio de Empleo y Seguridad Social:

Tipos de cotización (%)			
Contingencias	**Empresa**	**Trabajadores**	**Total**
Comunes	23,60	4,70	28,30
Horas Extraordinarias Fuerza Mayor	12,00	2,00	14,00
Resto Horas Extraordinarias	23,60	4,70	28,30

Tabla 55: Tipos de cotización 2015

Teniendo en cuenta estas dos tablas, se procede a calcular el coste de cotización durante el periodo que abarca el proyecto, es decir, el porcentaje de Seguridad Social que

[40] http://www.seg-social.es/Internet_1/Trabajadores/CotizacionRecaudaci10777/Basesytiposdecotiza36537/index.htm

paga la empresa. Para ello, se ha estimado la base cotizada, como dos veces la base mínima y una la base máxima, del grupo de cotización 2 de la *Tabla 54*, utilizando el tipo de cotización correspondiente. A continuación se muestran cada una de las cifras junto con el coste total de cotización:

Personal	Base cotizada (€)	Tipo (%)	Cotización/mes (€)	Meses dedicados	Coste total cotización (€)
Roberto Monsalve Toledo	1.786,40	23,60	421,60	7,75	3.267,40

Tabla 56: Coste de cotización

Como se puede comprobar en la *Tabla 56*, el coste total de cotización asciende a 3.267,40€. Por lo que, esta cifra sumada al coste de personal calculado en la *Tabla 53*, da como resultado el coste total de recursos humanos, que se muestra a continuación:

Coste total personal (€)	9.075,00
Coste total cotización (€)	3.267,40
COSTE TOTAL RECURSOS HUMANOS (€)	**12.342,40**

Tabla 57: Coste total recursos humanos

Como se puede observar en la *Tabla 57*, el coste total de recursos humanos asciende a 12.342,40€.

7.2.2 COSTES DE MATERIAL

A la hora de realizar el cálculo total asociado a los recursos materiales, se ha desglosado el coste por cada elemento utilizado, teniendo en cuenta la dedicación al proyecto, el precio de cada herramienta, su porcentaje de uso y el período de amortización. El cálculo se efectuará mediante la siguiente fórmula:

$$Coste = Precio\ elemento * Porcentaje\ de\ uso * \frac{Número\ meses\ dedicación}{Período\ de\ amortización}$$

A continuación, se incluye una tabla en la que se desglosa el coste de cada material, distinguiendo entre materiales hardware y software:

Descripción material	Tipo	Precio (€)	Porcentaje de uso	Meses dedicación	Período de amortización (meses)	Coste (€)
MacBook Pro Retina 13'' i5 2.5 GHz con 8GB	HW	1.349,00	100,00	7,75	60,00	174,25
Windows 8.1	SW	125,00	30,00	2	48	1,56
Office 365 Personal para Mac	SW	69,00	80,00	6,5	48	7,48
TOTAL						**183,29**

Tabla 58: Costes de material

Como se puede comprobar en la *Tabla 58*, los costes de material ascienden a 183,29€.

7.2.3 COSTES INDIRECTOS

Por lo que se refiere a los costes indirectos del proyecto, se ha tenido en cuenta el gasto en ADSL, el cual se detalla en la *Tabla 59*:

Descripción	Coste/mes (€)	Meses dedicación	Coste (€)
ADSL	25	7,75	193,75
TOTAL			**193,75**

Tabla 59: Costes indirectos

Como se comprueba en la *Tabla 59*, los costes indirectos ascienden a 193,75€.

7.2.4 COSTE TOTAL DEL PROYECTO

Por último, se incluye una tabla final en la que se suman los tres tipos de coste desglosados (recursos humanos, materiales e indirectos), se añade el beneficio esperado del proyecto y se aplica el IVA al importe final:

Descripción	Coste (€)
Costes totales de recursos humanos	12.342,40
Costes totales de material	183,29
Costes indirectos totales	193,75
Importe total sin beneficio	**12.719,44**
Beneficio esperado (20%)	2.543,89
Importe total con beneficio	**15.263,33**
IVA (21%)	3.205,29
PRESUPUESTO TOTAL	**18.468,62**

Tabla 60: Coste total del proyecto

A partir de la *Tabla 60*, se comprueba como el presupuesto total del presente Trabajo de Fin de Grado asciende a **18.468,62€ (Dieciocho mil cuatrocientos sesenta y ocho Euros con sesenta y dos céntimos)**.

8. CONCLUSIONES DEL TRABAJO REALIZADO Y TRABAJOS FUTUROS

8.1 CONCLUSIONES DEL TRABAJO REALIZADO

Llegado el final del presente Trabajo de Fin de Grado, se han alcanzado cada uno de los objetivos que fueron marcados al comienzo del mismo.

En primer lugar, se ha realizado una importante recopilación de los principales sistemas de almacenamiento basados en grafos, indicando las principales características de cada uno de ellos. Asimismo, se han analizado diferentes diseños en cuanto a la forma de almacenar la información en el grafo de Neo4j, escogiendo la solución que mejor se adaptaba al estudio.

Como parte más destacable del presente trabajo, se ha realizado el experimento entre Neo4j y el modelo RSHP, dando como resultado que Neo4j, con el diseño del grafo final adoptado, extrae mejor la información relevante, incluyendo al menos uno de los términos de la consulta, en cada uno de sus resultados.

Además, con este trabajo se provee un proceso automatizado que, a partir de un grafo creado en Neo4j, permite generar consultas de forma aleatoria (tanto en lenguaje Cypher como en lenguaje natural) y ejecutarlas programáticamente en Neo4j. Este proceso será muy útil para siguientes trabajos de investigación que requieran ejecutar una serie de consultas en Neo4j.

A nivel personal, este Trabajo de Fin de Grado me ha ayudado a mejorar distintas cualidades entre las que se encuentran la búsqueda y recopilación de información ya existente, cómo distinguir información relevante para el estudio de la que no lo es, cómo estructurar dicha información para el documento aquí expuesto, cómo resolver problemas inesperados para poder proseguir con el desarrollo del proyecto y la capacidad de análisis y síntesis.

Asimismo, en el presente estudio he podido aplicar algunos de los conocimientos adquiridos durante la carrera, como puede ser la realización de una planificación para un proyecto de tal envergadura, el presupuesto asociado al mismo o poder aplicar un modelo de desarrollo software.

En definitiva, he disfrutado con la realización de este estudio ya que las bases de datos es un tema que me apasiona. Además, he aprendido bastante sobre el mundo de las bases de datos orientadas a grafos, algo que no se incluye en el grado cursado y que me parece muy interesante y cada vez más importante para el ámbito profesional. Esto, unido al afán de investigación del proyecto, que también comparto yo, ha convertido este proyecto en un bonito reto para finalizar esta etapa como estudiante universitario.

8.2 TRABAJOS FUTUROS

Al tratarse de un ámbito muy amplio y de existir numerosos sistemas de almacenamiento basados en grafos, podrían incluirse numerosas líneas de investigación a partir de este trabajo. A continuación se exponen las más destacables para los futuros trabajos relacionados con este Trabajo de Fin de Grado:

- Se propone realizar el mismo estudio comparativo entre Neo4j y RSHP, pero aumentando considerablemente el volumen de datos indexado en cada uno de los sistemas (por ejemplo indexar 10.000 requisitos, en lugar de los 1.572 utilizados para esta comparativa). Con esto se quiere descubrir si el volumen de datos almacenados, afecta o no a la precisión y recall de las consultas para ambos sistemas.

- Se propone realizar la misma comparativa entre Neo4j y RSHP, pero indexando los datos a través de patrones también en KnowledgeMANAGER, para comprobar si de esa forma, mejoran los resultados obtenidos en RSHP.

- Se propone incluir otro sistema, que almacene RDF, al presente estudio. Se podría comparar Neo4j, RSHP y otro sistema con RDF (Virtuoso). De esta forma se ampliaría este Trabajo de Fin de Grado, utilizando otros lenguajes de consulta como SPARQL.

- Otras líneas de investigación futuras, podrían ser la comparativa entre Neo4j y RSHP con otros sistemas de almacenamiento basados en grafos. En concreto, se propone realizar pruebas de rendimiento, analizando la precisión y la recall, entre los siguientes sistemas:

    - Neo4j vs Sparksee
    - Neo4j vs RSHP vs GraphBase
    - Neo4j vs RSHP vs Infinite Graph

9. BIBLIOGRAFÍA

[1] J. Urbano, J. Morato, M. Marrero, and S. Sánchez-Cuadrado, "Recuperación y Acceso a la Información." [Online]. Disponible: http://ocw.uc3m.es/ingenieria-informatica/recuperacion-y-acceso-a-la-informacion/material-de-clase-1/01-IntroducciOn.pdf.

[2] "Database SQL Language Reference - Contents." [Online]. Disponible: https://docs.oracle.com/cd/E11882_01/server.112/e41084/toc.htm. [Visitada: 29-Jul-2015].

[3] "Data Integrity." [Online]. Disponible: http://docs.oracle.com/cd/E11882_01/server.112/e40540/datainte.htm. [Visitada: 29-Jul-2015].

[4] "Oracle/PLSQL: Joins." [Online]. Disponible: http://www.techonthenet.com/oracle/joins.php. [Visitada: 28-Jul-2015].

[5] "SPARQL 1.1 Query Language." [Online]. Disponible: http://www.w3.org/TR/sparql11-query/#sparqlSyntax. [Visitada: 29-Jul-2015].

[6] "Guía Breve de Web Semántica." [Online]. Disponible: http://www.w3c.es/Divulgacion/GuiasBreves/WebSemantica. [Visitada: 29-Jul-2015].

[7] "DBpedia." [Online]. Disponible: http://es.dbpedia.org/. [Visitada: 23-Jun-2015].

[8] "Intro to Cypher - Neo4j Graph Database." [Online]. Disponible: http://neo4j.com/developer/cypher-query-language/. [Visitada: 19-Apr-2015].

[9] "Online Training: Getting Started with Neo4j - Neo4j Graph Database." [Online]. Disponible: http://neo4j.com/graphacademy/online-course/. [Visitada: 04-Apr-2015].

[10] "El problema de los siete puentes de Königsberg | 7Puentes." [Online]. Disponible: http://7puentes.com/aboutus/konigsberg/. [Visitada: 04-Apr-2015].

[11] "Bases de datos orientadas a grafos y su enfoque en el mundo real | Washington Velásquez Vargas - Academia.edu." [Online]. Disponible: http://www.academia.edu/5731075/Bases_de_datos_orientadas_a_grafos_y_su_enfoque_en_el_mundo_real. [Visitada: 03-Apr-2015].

[12] I. Robinson, J. Webber, and E. Eifrem, *O'REILLY Graph Databases*, 2nd ed. .

[13] "Introducción a NoSQL - Graph DataBase Neo4j de Ariel Andres Nasca en Prezi." [Online]. Disponible: https://prezi.com/xaitejgkpqsv/introduccion-a-nosql-graph-database-neo4j/. [Visitada: 02-Aug-2015].

[14] "Use Cases - Neo4j Graph Database." [Online]. Disponible: http://neo4j.com/use-cases/. [Visitada: 03-Apr-2015].

[15] "Semantic Technologies AllegroGraph Triple Store RDF Web 3.0 Database,optimized SPARQL Query engine, Prolog and RDFS+ reasoner." [Online]. Disponible: http://franz.com/agraph/allegrograph/. [Visitada: 03-Apr-2015].

[16] "Introduction | AllegroGraph 5.0.1." [Online]. Disponible: http://franz.com/agraph/support/documentation/current/agraph-introduction.html. [Visitada: 04-Apr-2015].

[17] "Sparsity-technologies: Sparksee high-performance graph database." [Online]. Disponible: http://www.sparsity-technologies.com/. [Visitada: 30-Jul-2015].

[18] "Sparksee Technology overview." [Online]. Disponible: http://es.slideshare.net/SparsityTechnologies/sparksee-technology-overview. [Visitada: 30-Jul-2015].

[19] "The World's Most Powerful Graph DBMS." [Online]. Disponible: http://graphbase.net/MostPowerful.html. [Visitada: 01-Aug-2015].

[20] "Unrivalled Performance." [Online]. Disponible: http://graphbase.net/Performance.html. [Visitada: 01-Aug-2015].

[21] "Maximum Versatility." [Online]. Disponible: http://graphbase.net/Versatility.html. [Visitada: 01-Aug-2015].

[22] "Graph Engine Basics." [Online]. Disponible: http://www.graphengine.io/docs/manual/basics.html. [Visitada: 02-Aug-2015].

[23] "Trinity Specification Language." [Online]. Disponible: http://www.graphengine.io/docs/manual/TSL/index.html. [Visitada: 02-Aug-2015].

[24] B. Shao, H. Wang, and Y. Li, "Trinity: A Distributed Graph Engine on a Memory Cloud." [Online]. Disponible: http://research.microsoft.com/pubs/161291/trinity.pdf.

[25] "InfiniteGraph | Distributed Graph Database." [Online]. Disponible: http://www.objectivity.com/products/infinitegraph/. [Visitada: 02-Aug-2015].

[26] "Use Cases | Objectivity." [Online]. Disponible: http://www.objectivity.com/solutions/use-cases/. [Visitada: 04-Aug-2015].

[27] "HypergraphDB - A Graph Database." [Online]. Disponible: http://www.hypergraphdb.org/index. [Visitada: 04-Aug-2015].

[28] "HyperGraphDB Data Management for Complex Systems." [Online]. Disponible: http://www.hypergraphdb.org/docs/HyperGraphDB-Presentation.pdf.

[29] "OrientDB - OrientDB Multi-Model NoSQL DatabaseOrientDB Multi-Model NoSQL Database." [Online]. Disponible: http://orientdb.com/orientdb/. [Visitada: 04-Aug-2015].

[30] "Why OrientDB? - OrientDB Multi-Model NoSQL DatabaseOrientDB Multi-Model NoSQL Database." [Online]. Disponible: http://orientdb.com/why-orientdb/. [Visitada: 04-Aug-2015].

[31] "OrientDB vs Neo4j - OrientDB Multi-Model NoSQL DatabaseOrientDB Multi-Model NoSQL Database." [Online]. Disponible: http://orientdb.com/orientdb-vs-neo4j/. [Visitada: 05-Aug-2015].

[32] "The List of Featured Graph Database Overviews and Benchmarks - Blog on All Things Cloud Foundry." [Online]. Disponible: http://blog.altoros.com/the-list-of-featured-graph-database-overviews-and-benchmarks.html. [Visitada: 05-Aug-2015].

[33] "Google Search Statistics - Internet Live Stats." [Online]. Disponible: http://www.internetlivestats.com/google-search-statistics/. [Visitada: 23-Aug-2015].

10. ANEXOS

10.1 ANEXO A: EXTENDED ABSTRACT

10.1.1 INTRODUCTION

This chapter contains a brief introduction to this Final Degree Work. It will be exposed the existing problem, the motivation to carry out the work and objectives to be achieved at the end of it.

10.1.1.1 EXISTING PROBLEM

For a long time relational databases have been the most used databases by many companies to store and manage their data. However, in recent times, information technologies and the new data-based environment have rapidly evolved, forcing organizations to adapt their management technology to meet new challenges. This new context has been driven by a significant increase in the volume of data that must be handled every day.

With these new challenges appears the concept of Big Data, which refers to the large set of complex data that is difficult to process by traditional systems. Mainly, Big Data is characterized by its famous "three Vs": variety (different data types, structured and unstructured), volume (high scalability, thousands of nodes) and speed (data is processed on servers where they are). Along with this concept, new management systems, called NoSQL databases, appear as an alternative to classical relational databases management systems. These new NoSQL systems are characterized, as its name suggests, by not using SQL as the primary query language (some NoSQL systems support it), besides they do not require fixed structures such as tables to store data.

In general, NoSQL databases use one of the following data models: model based on documents, graphs or key-value data model. This Final Degree Work is conceived with an enthusiasm for researching databases based on graphs. In addition to an analysis of these databases and to review the characteristics of the most important management systems, this document also shows a comparison between the RSHP model (using the KnowledgeMANAGER tool) and Neo4j to establish which of the two systems extracts information, for specific terms, in the best way.

10.1.1.2 WORK MOTIVATION

The motivation for this study stems from curiosity to investigate databases based on graphs. Those databases are increasingly rising in different professional fields. This means that this project helps to understand where these databases are today and how they may become important for companies. The research provides an opportunity to explore the graph-based databases area and study the current alternatives available in the market.

10.1.1.3 OBJECTIVES

The main objectives of this work emerges from the motivation and are listed below:

- Analyse of the different alternatives on graphs databases management systems that are available in the market.

- Make a comparison between Neo4j and RSHP model (through KnowledgeMANAGER tool) as alternatives for information retrieval. Some queries, with concrete terms, will be executed to know which of the two systems gets relevant information in a best way to each of them. Precision and recall measures are then calculated for each query to analyse the performance of each information retrieval engine.

- Study of the different mechanisms to store information in a Neo4j graph with the aim of making a comparison of the impact in the information retrieval process.

- Implement a query generation module and Neo4j query execution automatic process. Queries will be generated from the Neo4j graph, in Cypher language to Neo4j and in natural language to RSHP. Natural language queries will be executed in KnowledgeMANAGER tool, while the ones in Neo4j will be executed via the Neo4j API.

10.1.2 DESCRIPTION OF THE UNDERLYING SYSTEMS: NEO4J and KnowledgeMANAGER

This chapter contains a summary description of the two systems involved in the comparison, Neo4j and KnowledgeMANAGER (which follows RSHP model).

10.1.2.1 NEO4J

Neo4j is an open source database based on graphs, written in Java, which can store data in a structured way. It integrates perfectly with other languages like PHP, Ruby, .Net, Python, Node and Scala. The database is embedded in a Jetty server (http server 100% Java based). Neo4j is compatible with Linux, Windows and Mac OS X. The Neo4j query language is Cypher, a graph query language. Social networks and recommendation systems are ideal fields for working with Neo4j.

There are two versions of Neo4j, Community Edition (open source) and Enterprise Edition. To perform concept tests the free version it is enough but if we want take advantage of Neo4j, we have to use the EE version, which allows us to perform monitoring, hot system backups and high performance cache, among other advantages.

Neo4j has no scheme and transactions are ACID (Atomicity, Consistency, Isolation and Durability). The data model that uses Neo4j is a Property Graph. This type of graph is composed of nodes labelled and routed relations, all with their respective properties.

There are two ways to use Neo4j in applications: embedded in applications or working as a server. Some features of each are the following:

- **Neo4j embedded in an application:** you can include Neo4j libraries in an application to take advantage of its characteristics without having to rely on an external server. This option is similar to work with in-memory database. Embed the database in an application can also be done in two different ways. Depending on performance needs and the resources we have, it will be chosen one of these two alternatives. A description of each of them is done now:

    1. **Use a single instance of Neo4j:** if you are in an environment with few resources (memory, CPU, etc.) or application requirements are not too high, this option will be the best. This simple instance is accessed via GraphDatabaseService API.

    2. **Use multiple instances:** when you need high data availability and you have many resources, the best option is HighlyAvailableGraphDatabase API, which make easier the creation of multiple instances to get your goals.

- **Neo4j working as a server:** is the most common way to access a database. It works like a typical database server, running in one or more machines (depending on the needs), which handles customer requests. In this case, REST requests are sent to the server to resolve queries.

10.1.2.1.1 ADVANTAGES AND DISADVANTAGES

Some of the main Neo4j advantages are listed below:

1. Neo4j provides the ability to efficiently manage large amounts of data. More specifically, its capacity is about 34.000 million nodes, 34.000 million relationships, 68.000 million properties and 32.000 types of relationships.

2. Similarly, one of its main advantages is the high-speed query execution. It can skip million times between nodes in just a few seconds. This is possible thanks to the graph processing natively. So the storage is optimized for queries based on close data, rather than global queries to the database.

3. Another advantage is its flexible data model, which allows you to create nodes and relationships without needing to declare the data type of them.

4. In addition, Neo4j is an open source, which means it is easily accessible to everybody.

5. You can execute queries through a REST API, which facilitates integration with Web applications.

6. Another advantage is how Neo4j stores nodes, relationships and properties. Perform a Graph Native Storage. This means that Neo4j stores nodes in a file, in another file it stores relationships and properties of both in other one. So, having properties in another file, nodes and relationships storage are concerned exclusively with the structure of the graph. In this way, you can quickly get nodes, based on their id, into memory, because you know what position are these. This kind of storage lets Neo4j provide high performance when you run queries on large amount of data.

The main disadvantages of Neo4j are:

1. Compatibility Issues: NoSQL databases have few standards in common.
2. Neo4j has not user management.
3. Lack of experience: it is not mature enough for some companies.
4. Neo4j Community does not scale to very large data sets.
5. Neo4j prefers to have the entire graph into memory, but this is unnecessary and inefficient. The best option is to keep in memory only the data set that will be asked.
6. If you store too much information on nodes and the problem has not modelled well, queries performance will fall considerably, becoming inefficient, due to the nodes storage take up too much memory.

10.1.2.1.2 USE CASES

This section contains the main use cases in which Neo4j operates vey well:

- Master Data Management
- Network and IT Operations
- Real-Time Recommendations
- Fraud Detection
- Social Network
- Identity and Access Management
- Graph-Based Search

10.1.2.2 KNOWLEDGEMANAGER

KnowledgeMANAGER (KM) is a tool whose main purpose is the ontologies management. It is commercial product created by The European information technology company called “The Reuse Company". Members of Carlos III University of Madrid have taken part on the development of this tool, more specifically some members of the Knowledge Reuse Group within the Department of Computer Science. KM includes a set of features among which are:

- Language issues
- Vocabulary Management
- Relationships
- Patterns
- Thesaurus

Nowadays, knowledge is the most valuable asset for modern organizations. Proper management of the organization of knowledge is the key to success. Knowledge must be obtained from numerous sources and stored in a secure repository. In this way, it will become a reusable component for software projects.

KM is focused precisely on knowledge management. It allows creating knowledge to be reused later relying on different concepts such as creating vocabularies, tokenization and normalization of terms, syntactic labels, disambiguation of terms and patterns.

10.1.2.2.1 RSHP MODEL

KM is based on RSHP model. This model joined with Neo4j, are the two main elements of this study. The RSHP universal knowledge representation model is based on the idea that any information can be described by a set of relationships between different concepts. With this approach, the main element of an information unit is the relationship. For example, the data model entity-relationship is represented as relations between different entity types or software objects models that can be represented by relationships between objects or classes.

RSHP model includes a repository to store information and relationships in order to reuse all kinds of knowledge. The natural language may also be represented by links between various terms, using the same structure as in the previous examples. Specifically, to represent the natural language sentences should be used under the Subject + Verb + Predicate structure (SVP), which can be regarded as a relationship (V) between subject (S) and predicate (P). Mainly, RSHP is based on the following principles:

- The main description element is the relationship, which is the element in charge of linking knowledge elements.

- A knowledge element (KE) is an atomic knowledge component that appears on an artifact that is connected by means of one or more relationships with other KEs to build information. It is defined by a concept, which is represented by a normalized term. Artifacts are containers of KEs and their relationships.

In RSHP, the representation model to describe the contents of any type of artifact (requirements, risks, models, tests, text documents or source code) should be performed by RSHPs where each RSHP is called "RSHP-description" and described by KEs. An important feature of this model is that there is no restriction to represent a particular type of knowledge. In addition, RSHP has been used as a model for building information indexing and retrieval systems for general-purpose, domain representation models, the evaluation of quality requirements and knowledge management tools as KnowledgeMANAGER.

In conclusion, the main components of this RSHP model are listed below:

- **Artifact:** it is a container of relationships and knowledge elements.
- **RSHP:** is a relationship between different elements of knowledge.
- **KnowledgeElement (KE):** is a knowledge element. It can also be identified as a term occurrence.
- **Term:** is a string or literal.
- **TermTag:** represents the syntactic category of each term.
- **MetaProperty:** represents properties owned by each KE.
- **SemanticCluster:** involved in creating RSHP relationships between different terms.

10.1.3 DESIGN OF THE NEO4J GRAPH

Firstly, it must be noticed that data used to the comparison is a set of requirements. Before indexing requirements in the Neo4j graph, it is necessary to think about what information would be store on the graph and how to do that to perform the study successfully. From the beginning, it was decided that the text of each requirement and its corresponding identifier, would be stored. No further information was stored as it was irrelevant to the experiment.

After analysing different alternatives, this section contains the description of the Neo4j graph. In the proposed final solution, the graph is created by patterns. Through them, the requirements are indexed in the graph. Note that each pattern can index a large number of requirements, so too patterns are not needed to complete the graph used in the experiment.

It is important to note two important aspects of patterns. First, patterns have certain terms, which are fixed, and others may be variable. That is, the pattern fixed terms must be mapped directly to the requirement terms to be indexed properly. So, if the second position of the pattern has the fixed term “system”, the same requirement position must have the same term. However, variable terms can match with the requirement term or any other, so accept any term in that position of the pattern. Variable terms will appear between brackets.

The other aspect to highlight is related to the identification of what terms will be nodes and what will be relationships. It is necessary to specify this issue to patterns, so that when the graph is being created, they know what terms are nodes and what are relationships. This has been decided to mark on the patterns, with a parentheses "r" (relationship) or "n" (node) before each term. A valid pattern could look as the following pattern:

Pattern: (n)The (n)[system] (r)must (n)[action]

Figura 143: Pattern example

So both nodes and relationships store terms, taking the "id" and "requirements" properties. It is important to explain that not all relationships have these two properties because if a pattern (*Figure 143*) has two consecutive terms marked as nodes, an empty relationship between them is created and that relationship does not store any term.

10.1.4 AUTOMATIC QUERY GENERATION MODULE

This work also includes an automatically process to generate queries which will be executed in both systems. It should be noted that queries are generated in two ways. It is generated Cypher queries for Neo4j and natural language queries for RSHP. This part has been included to speed up the process of creating queries to be executed. This way, no extra time is wasted to make hand-made queries.

Terms used in queries are chosen randomly from the vocabulary of the graph in Neo4j. Besides, it does not support the same term in a query and any term is repeated until all of them have appeared in previous queries.

So the final solution of the automatic generation of queries is as follows:

1. Vocabulary of the graph is collected via two queries, one to terms stored in nodes and another to terms stored in relationships.

2. Then, it begins to create queries the user has indicated. Terms are selected randomly from the recovered vocabulary, for each query. It was decided that each query would be between 1 and 3 terms due to the fact that is a common number of terms used in well-known search engines. Once terms are selected, queries are built for both systems.

3. When a term stored on a node and a term stored on relationships is selected to the same query, it will be divided into two queries. This is done to recover requirements correctly. Note that both types of queries, the OR operator is used when the query includes more than one term.

10.1.5 NEO4J AUTOMATIC QUERY EXECUTION

As explained in the previous section, Neo4j and RSHP queries are generated. RSHP queries are executed by hand, but the ones in Neo4j are executed programmatically. This way, an automatic query execution environment is created in Neo4j saving a lot of time. Many queries can be executed only once, clearly separating the results of each one.

For Neo4j automatic query execution, a method has been implemented, in which two parameters are given. The first parameter corresponds to the graph on which the queries are executed and the second one includes the file path, where Cypher queries are. After executing queries file, results of each query are included into a different line of another file.

10.1.6 EXPERIMENTATION

10.1.6.1 EXPERIMENT DESIGN

1. Define the dataset to use.
2. Create the Neo4j graph.
3. Indexing requirements into KnowledgeMANAGER.
4. Define and create a set of queries.
5. Identify relevant results to each query.
6. Run queries in KnowledgeMANAGER and Neo4j.
7. Calculate the precision and recall thrown by each system for each query.
8. Analyse which of the two systems provides better results for the experiment.

10.1.6.2 EXTENSION OF THE EXPERIMENT DESIGN

As already mentioned in previous chapters, this study aims at performing a comparison between RSHP and Neo4j to know how relevant is the information retrieved. To this study, relevant results are those requirements that contain at least one of query terms.

Firstly, it should be noted that data used for this study is a requirements set, specifically, 1572 requirements written in English, indicating different actions for different users in different situations. This data has been indexed in Neo4j as it explained previously.

To perform the indexing process in KnowledgeMANAGER, each requirement has been included in a separate file, thus creating 1.572 files stored in a folder. This folder is the one that has been indexed in KM. After indexing data in KM, artifacts must be created. An artifact is created for each indexed requirement. An artifact contains requirement terms with semantic relevance. Thus, KM creates a formal representation for each artefact that includes those terms with semantic significance. This will affect negatively queries that include any of these terms not include in the formal representation.

After having indexed data in both systems, 30 queries have been generated between 1 and 3 terms randomly, using the mechanism described in previous chapters. Each query have been generated both natural language (for RSHP) and Cypher (for Neo4j). In section *Results Obtained* results of each system are listed.

Queries written in Cypher are programmatically executed in Neo4j. RSHP execution queries can be done in two different ways. KM allows searches by inclusion and similarity. In this experiment, the used method is search by inclusion, since the objective is to recover artifacts that include query terms.

It should be pointed out that, for the study, it will only consider the first 20 results returned by each system, for each query. So, it is going to make a cut-off for the first 20 results. Also, it remembers that the goal of this study is to determine which of the two systems gets relevant requirements containing any query term in a better way.

10.1.6.3 MAIN RESULTS AND DISCUSSION

In this section the results of executing 30 queries in KM and Neo4j are presented in the table below:

Query Identifier	Neo4j			RSHP		
	Precision	**Recall**	**F1**	**Precision**	**Recall**	**F1**
q1	1	0,07	0,13	1	0,03	0,06
q2	1	0,42	0,59	0,25	0,1	0,14
q3	0,8	0,12	0,21	1	0,15	0,26
q4	1	0,13	0,23	0	0	0
q5	1	0,83	0,91	0	0	0
q6	1	0,01	0,02	1	0,01	0,02
q7	1	0,01	0,02	0,9	0,01	0,02
q8	0,75	0,01	0,02	1	0,01	0,02
q9	1	0,01	0,02	0	0	0
q10	1	0,05	0,1	1	0,05	0,1
q11	1	0,02	0,04	1	0,02	0,04
q12	1	0,12	0,21	1	0,07	0,13
q13	1	0,15	0,26	1	0,15	0,26

q14	0,9	0,11	0,2	1	0,12	0,21
q15	1	0,13	0,23	0	0	0
q16	1	0,83	0,91	0,4	0,33	0,36
q17	1	0,01	0,02	1	0,01	0,02
q18	1	0,06	0,11	1	0,06	0,11
q19	0,8	0,01	0,02	1	0,01	0,02
q20	1	0,06	0,11	1	0,06	0,11
q21	1	0,09	0,17	1	0,09	0,17
q22	0,9	0,12	0,21	0	0	0
q23	1	0,01	0,02	1	0,01	0,02
q24	1	0,06	0,11	1	0,06	0,11
q25	1	0,15	0,26	1	0,15	0,26
q26	0,8	0,04	0,08	1	0,03	0,06
q27	1	0,01	0,02	1	0,01	0,02
q28	1	0,03	0,06	1	0,03	0,06
q29	1	0,05	0,1	1	0,05	0,1
q30	1	0,13	0,23	0,8	0,1	0,18
Average	**0,97**	**0,13**	0,23	**0,78**	**0,06**	0,11

Tabla 61: Neo4j and RSHP results (Precision, Recall and F-measure)

From *Table 61*, it is checked as Neo4j maintains a constant precision, always between 80% and 100% precision. In other words, Neo4j introduces just a little noise in its results. These small drops in the precision of some Neo4j queries, is caused by the accumulation of terms in nodes. However, RSHP precision is more variable because, in 9 of the executed queries, requirements that were obtained were not relevant to the query. That is, RSHP introduces more noise than Neo4j, which causes lower precision.

In addition, Neo4j queries recall is, in general, better than RSHP. In some queries, recall is the same for both systems, but Neo4j stands on RSHP in other queries as query 2, 5 and 16. It should be noted that many queries have low recall values for both Neo4j to RSHP due to the high number of relevant requirement for each query. **So, RSHP results have more silence than Neo4j results. The lower silence is, the more relevant results are recovered.** In this sense, Neo4j provides better results.

The main conclusion of this study is that Neo4j, in general, produces better results than RSHP model, although to three terms queries, RSHP results improve significantly, being more similar to those obtained in Neo4j. As seen, Neo4j is great to recovery requirements according to concrete terms, maintaining an excellent precision in almost all queries. However, RSHP may improve its measures of precision and recall.

Therefore, it is evident that Neo4j operates better with queries on specific terms than RSHP model, which gets greater potential for intelligent queries. That is, RSHP has a greater potential for more general queries, in which results are relevant not only those containing a query term. Thus, acquired knowledge application can offer more interesting results. In this case, RSHP applying knowledge obtained (for example, through clustering), which causes recover not relevant results to this study.

10.1.7 CONCLUSIONS AND FUTURE WORK

10.1.7.1 CONCLUSIONS

Come the end of this Final Degree Work, it has been achieved each objective that were set out at the beginning of it.

Firstly, there has been a significant collection of major storage systems based on graphs indicating the main features of each one. Furthermore, it has been analysed the different designs regarding how to store information in the Neo4j graph, choosing the best solution to the study.

As one of the main outcomes of this work, it has performed the experiment between Neo4j and RSHP model, resulting that Neo4j extracts relevant information in a better way, including at least one of the query terms in each of its results.

In addition, this work provides an automated process that can generate random queries (both Cypher language and natural language) and execute programmatically in Neo4j. This process will be very useful for subsequent researches that require executing series of queries in Neo4j.

On a personal level, this Final Degree Project has helped me improve different qualities which are the pursuit and collection of existing information, how to distinguish relevant information for the study from which it is not, how to structure information to the document here exposed, how to solve unexpected problems in order to continue the project and the ability of analysis and synthesis.

Besides, in this study I have applied some of the knowledge acquired during the degree, such as performing a project schedule, the associated budget of it or to apply a software development model.

Finally, I have enjoyed the carrying out of this study because databases world is a matter that fascinates me. Also, I have learned a lot about graph databases, which is not included in the degree subjects and which I find very interesting and increasingly important area for professional field. This project has become a nice challenge to finish this stage as a university student.

10.1.7.2 FUTURE WORK

The issue analysed in this work belongs to a very large scope and there are a lot of systems based on graphs. Taking this into account there are many future lines to work. The most important lines in order to investigate in the future are the following ones:

- It is proposed to carry out a comparative study between Neo4j and RSHP model with a considerably increasing of the data volume indexed in each system (e.g. 10.000 requirements, rather than the 1.572 used for this comparison). With this, it wants to check whether the volume of stored data affects the precision and recall of queries for both systems.

- It is proposed to make the same comparison between Neo4j and RSHP model indexing data through patterns in KnowledgeMANAGER to see if that way, improve its results.

- It is proposed to add another system, which stores RDF, to the present study. It could compare Neo4j, RSHP model and a system with RDF (Virtuoso). Thus, this Final Degree Work is extended using other query languages like SPARQL.

- Another future research could be a comparative between Neo4j and RSHP with other graph storage systems. Specifically, it is proposed performance testing, analysing precision and recall, among the following systems:

    - Neo4j vs Sparksee
    - Neo4j vs RSHP vs GraphBase
    - Neo4j vs RSHP vs Infinite Graph

10.2 ANEXO B: MANUAL REDUCIDO DE SQL

10.2.1.1 LENGUAJE DE DEFINICIÓN DE DATOS (DDL)

En este apartado se detalla un ejemplo de creación de una tabla con SQL, a través del cual, se profundiza en otros temas importantes como pueden ser las restricciones de columna, de tabla y las distintas acciones referenciales permitidas. A continuación se incluye dicho ejemplo:

```
CREATE TABLE "ACTOR"
  (       "ID_ACTOR" NUMBER NOT NULL,
          "NOMBRE" VARCHAR2(20 BYTE) NOT NULL,
          "APELLIDO1" VARCHAR2(20 BYTE) NOT NULL,
          "APELLIDO2" VARCHAR2(20 BYTE),
          "AÑO_NACIMIENTO" NUMBER NOT NULL,
          "SEXO" VARCHAR2(1 BYTE) NOT NULL,
           CONSTRAINT "ACTOR_PK" PRIMARY KEY ("ID_ACTOR"),
           CONSTRAINT "ACTOR_CK" CHECK (SEXO IN ('H','M'))
  );
```

Figura 144: Creando tabla "ACTOR" con SQL en Oracle

A partir de la *Figura 144*, se comprueba cómo se usa la sentencia CREATE para crear una tabla con SQL, indicando entre paréntesis los campos que contendrá junto con sus respectivos tipos de datos. De la misma forma se especifica si el campo admite valores nulos o no.

Después de declarar todos los campos, se indica cuál es la clave primaria de la tabla (en este caso *ID_ACTOR*), algo indispensable ya que no puede haber ninguna tabla sin clave primaria. Como se puede observar se utiliza la cláusula CONSTRAINT para dar un nombre a la restricción *PRIMARY KEY*. Esto no es estrictamente necesario pero es conveniente nombrar cada una de las restricciones de cada tabla, para que la definición sea lo más clara posible. Por último, se aplica la restricción *CHECK* para indicar que los valores que tomará el campo *SEXO* serán "H" o "M". Es decir, en la tabla *ACTOR*, sólo se admitirán tuplas que incluyan en el campo *SEXO* una "H" o una "M". En el caso de intentar insertar una fila con otro valor en ese campo, la inserción fallaría.

En la figura anterior, se han declarado dos restricciones denominadas "restricciones de tabla" (*ACTOR_PK* y *ACTOR_CK*) que siempre se deberán cumplir. Asimismo, se han declarado cinco "restricciones de columna" *NOT NULL*, las cuales también se deberán cumplir siempre. Las restricciones de columna, como su propio nombre indica, sólo afectan al campo en el que se incluyen y se escriben en la misma línea que la definición del campo, mientras que las restricciones de tabla, afectan a toda la tabla y se especifican después de haber definido todos los campos de la tabla. A continuación se muestran los distintos tipos de restricciones de columna y de tabla que se pueden aplicar [3].

Restricciones de columna	
Restricción	**Descripción**
NOT NULL	La columna no admite valores nulos.
UNIQUE	La columna no puede tener valores repetidos ya que se trata de una clave alternativa.
PRIMARY KEY	La columna no puede tener valores nulos ni repetidos ya que es la clave primaria.
REFERENCES tabla (columna)	La columna es la clave ajena de la columna de la tabla especificada.
CHECK (condiciones)	La columna debe cumplir las condiciones especificadas.

Tabla 62: Restricciones de columna - SQL

Restricciones de tabla	
Restricción	**Descripción**
UNIQUE (columna [, columna..])	El conjunto de columnas especificadas no puede tener valores repetidos ya que se trata de una clave alternativa.
PRIMARY KEY (columna [, columna…])	El conjunto de columnas especificadas no puede tener valores nulos ni repetidos ya que se trata de la clave primaria.
FOREIGN KEY (columna [, columna…]) REFERENCES tabla (columna2 [, columna2…])	El conjunto de las columnas especificadas es una clave ajena que referencia la clave primaria formada por el conjunto de columnas2 de la tabla indicada.
CHECK (condiciones)	La tabla debe cumplir las condiciones especificadas.

Tabla 63: Restricciones de tabla - SQL

Como se aprecia en las dos tablas anteriores, algunas restricciones aparecen en ambas. Esto es debido a que si la restricción sólo afecta a una columna, se puede incluir en la misma línea de definición del campo, mientras que si la restricción afecta a varias columnas, se debe incluir una restricción de tabla especificando todos los campos afectados. Aunque una restricción sólo afecte a una columna, se puede añadir restricción de tabla (En la *Tabla 63* los corchetes indican que esa parte es opcional). Además, cabe destacar que todas estas restricciones vistas son denominadas **restricciones de integridad** ya que son las encargadas de mantener la consistencia semántica de los datos.

En el caso de las restricciones **REFERENCES tabla (columna)** y **FOREIGN KEY (columna [, columna…]) REFERENCES tabla (columna2 [, columna2…])** es importante indicar qué acciones tomar cuando se modifique o se borre la clave referenciada. En este sentido, se indica a continuación las cláusulas a utilizar.

```
FOREIGN KEY (clave_ajena) REFERENCES tabla (clave_primaria)
[ON UPDATE {NO ACTION | CASCADE | RESTRICT | SET DEFAULT | SET NULL}]
[ON DELETE {NO ACTION | CASCADE | RESTRICT | SET DEFAULT | SET NULL}]
```

Figura 145: Restricción de integridad referencial - SQL

Como se puede observar en la *Figura 145*, mediante **ON DELETE acción** y **ON UPDATE acción**, se especifica qué hacer en el caso de que se modifique o borre la clave referenciada. Las acciones que se pueden adoptar son **NO ACTION, CASCADE, RESTRICT, SET DEFAULT y SET NULL**, las cuales se procede a explicar en la siguiente tabla.

Acciones referenciales	
Acción	**Descripción**
CASCADE	Cada vez que se eliminan o actualizan las tuplas de la tabla referenciada, se eliminan o actualizan las filas de la tabla desde donde se referencia.
RESTRICT	El valor de una columna de la tabla referenciada no puede ser modificado o eliminado si en la tabla donde está la clave ajena, existe una fila que hace referencia a ese valor de la columna de la tabla referenciada.
NO ACTION	NO ACTION y RESTRICT son muy parecidos. La diferencia entre ambos es que NO ACTION efectúa la comprobación de integridad referencial después de intentar ejecutar UPDATE o DELETE, mientras que RESTRICT realiza la comprobación antes. Si la restricción de integridad no es satisfactoria (se pierde la consistencia semántica de los datos), se obtendrá un error.
SET NULL	Cuando se modifican o eliminan filas de la tabla referenciada, se actualizan o eliminan las filas de la tabla desde donde se referencia, poniendo a NULL los campos correspondientes.
SET DEFAULT	Cuando se modifican o eliminan filas de la tabla referenciada, se actualizan o eliminan las filas de la tabla desde donde se referencia, poniendo el valor por defecto que se haya asignado a los campos correspondientes.

Tabla 64: Acciones referenciales de la restricción de integridad referencial - SQL

Siguiendo con el ejemplo de la tabla anterior, se crea una nueva tabla *PELÍCULA* y después, para almacenar las películas en las que ha actuado cada actor, se crea otra tabla *PELÍCULA_ACTOR*, ya que es una relación N:M. Con esto, se presenta un ejemplo en el que existen dos restricciones de integridad referencial, en el que se indicará qué acción realizar en caso de borrado de los datos de las tablas referenciadas.

```
CREATE TABLE "PELÍCULA"
  (       "ID_PELÍCULA" NUMBER CONSTRAINT "PELÍCULA_PK" PRIMARY KEY,
          "TÍTULO" VARCHAR2(50 BYTE) NOT NULL,
          "DIRECTOR" VARCHAR2(25 BYTE) NOT NULL
);

CREATE TABLE "PELÍCULA_ACTOR"
  (       "ID_ACTOR" NUMBER NOT NULL,
          "ID_PELÍCULA" NUMBER NOT NULL,
          "ROL" VARCHAR2(20 BYTE) NOT NULL,
           CONSTRAINT "PELÍCULA_ACTOR_PK" PRIMARY KEY ("ID_ACTOR", "ID_PELÍCULA"),
           CONSTRAINT "ACTOR_FK" FOREIGN KEY ("ID_ACTOR")
           REFERENCES "ACTOR" ("ID_ACTOR") ON DELETE CASCADE,
           CONSTRAINT "PELÍCULA_FK" FOREIGN KEY ("ID_PELÍCULA")
           REFERENCES "PELÍCULA" ("ID_PELÍCULA") ON DELETE CASCADE
  );
```

Figura 146: Ejemplo de creación de tabla con claves ajenas en Oracle

En la *Figura 146* se puede comprobar cómo se definen las claves primarias de forma distinta en cada una de las dos tablas, como se ha visto anteriormente en las *Tablas 62 y 63*.

Asimismo, se pueden observar las dos claves ajenas existentes en la tabla *PELÍCULA_ACTOR*, una de ellas referenciando a la tabla *ACTOR* y la otra a la tabla *PELÍCULA,* ambas con borrado en cascada. Cabe destacar que los tipos de datos de las claves ajenas deben coincidir con el tipo de datos de los campos referenciados. En caso contrario, la creación de la tabla fallará.

10.2.1.2 LENGUAJE DE MANIPULACIÓN DE DATOS (DML)

En este apartado se van a incluir algunos ejemplos de las cláusulas SELECT, INSERT, UPDATE y DELETE.

- **SELECT:** a continuación se incluyen algunos ejemplos utilizando esta cláusula:

```
SELECT *
FROM ACTOR
WHERE NOMBRE = "Keanu" AND APELLIDO1 = "Reeves";
```

Figura 147: Consulta SQL - Recuperando los datos de Keanu Reeves

Como se puede observar en la *Figura 147*, no se han especificado los campos que desean recuperarse, sino que se ha incluido un asterisco (*), lo cual significa que se quieren obtener todas las columnas pertenecientes a la fila de Keanu Reeves en la tabla *ACTOR*.

```
SELECT NOMBRE, APELLIDO1, AÑO_NACIMIENTO
FROM ACTOR
WHERE AÑO_NACIMIENTO > 1950
ORDER BY AÑO_NACIMIENTO
LIMIT 25;
```

Figura 148: Consulta SQL - Actores nacidos a partir de 1951

Con la consulta que aparece en la *Figura 148*, se recupera el nombre, primer apellido y año de nacimiento de aquellos actores que hayan nacido a partir del año 1951, ordenándolos de forma ascendente por el año de nacimiento. Además, con la cláusula LIMIT se recuperan sólo los 25 primeros, es decir, los 25 más veteranos.

```
SELECT ID_ACTOR, COUNT(*) AS "NÚMERO PELÍCULAS"
FROM PELÍCULA_ACTOR
GROUP BY ID_ACTOR
HAVING COUNT(*) > 2;
```

Figura 149: Consulta SQL - Actores que hayan actuado en más de 2 películas

Como se puede apreciar en la *Figura 149*, se agrupan las filas de la tabla *PELÍCULA_ACTOR* por el campo *ID_ACTOR* y sólo se recuperan aquellos actores que aparezcan en más de dos filas, es decir, que hayan actuado en al menos 3 películas. Cabe destacar que en SQL, se puede asignar un "alias" a las columnas que se recuperan en una consulta mediante la palabra clave "AS", seguida del nombre que se quiera dar entre comillas. Como se puede comprobar en la *Figura 149*, a la función de agregación *COUNT* que se devuelve, se le asigna un nombre con semántica para que la lectura de los datos sea más sencilla e inmediata.

- **INSERT:** en la siguiente figura se incluye un ejemplo de inserción en la tabla *PELÍCULA*.

```
INSERT INTO PELÍCULA (TÍTULO, DIRECTOR)
VALUES ("TITANIC","James Cameron");
```

Figura 150: Ejemplo de inserción sobre la tabla "PELÍCULA"

Como se puede observar en la *Figura 150*, se realiza una inserción de un registro en la tabla *PELÍCULA*, indicando el título y el director ya que ambos campos son obligatorios. Cabe destacar que no se incluye el campo *ID_PELÍCULA* que es la clave primaria de la tabla. Esto es debido a que este campo será auto-incremental por lo que no se incluye en la inserción, ya que se inserta automáticamente cuando se ejecuta la sentencia de la *Figura 150*.

- **UPDATE:** un ejemplo de uso de esta sentencia podría ser el que se indica en la siguiente figura.

```
UPDATE ACTOR
SET AÑO_NACIMIENTO = 1964
WHERE NOMBRE = "Keanu" AND APELLIDO1 = "Reeves";
```

Figura 151: Ejemplo de actualización sobre la tabla "ACTOR"

Como se puede observar en la *Figura 151*, se actualiza el campo *AÑO_NACIMIENTO* de la tabla *ACTOR*, para el registro cuyo nombre es *Keanu* y cuyo apellido es *Reeves*.

- **DELETE:** si se quisieran eliminar todos aquellos actores nacidos antes del año 1951, se debería ejecutar la siguiente sentencia.

```
DELETE FROM ACTOR
WHERE AÑO_NACIMIENTO < 1951;
```

Figura 152: Eliminando registros de la tabla "ACTOR"

10.2.1.3 OTRAS SENTENCIAS EN SQL

En este apartado se incluyen cuatro sentencias importantes a la hora de trabajar con una base de datos. Las dos primeras son **COMMIT y ROLLBACK**, que permiten confirmar y deshacer los cambios realizados en la base de datos, respectivamente. Las otras dos son **GRANT Y REVOKE**, las cuales permiten otorgar y eliminar permisos respectivamente, a los usuarios para poder acceder a los objetos de la base de datos que se deseen.

10.2.1.3.1 COMMIT Y ROLLBACK

Antes de detallar estas dos sentencias, es fundamental conocer el término **transacción**. Una transacción es un conjunto de sentencias SQL que es tratado como una única unidad lógica de trabajo. Es decir, un conjunto de sentencias SQL que se ejecuta como una única sentencia.

De la misma forma, es importante saber que cuando un usuario se conecta a una base de datos, se crea una **instancia** de la base de datos para él. Todas las transacciones que ejecute el usuario se aplicarán en su instancia pero no se harán efectivos en la base de datos.

Con todo esto, la sentencia **COMMIT** permite a un usuario confirmar de forma explícita todos los cambios realizados en una transacción. Es decir, esta sentencia permite que los cambios de la transacción sean permanentes en la base de datos, no sólo en la instancia del usuario. La forma básica de ejecutar esta sentencia es simplemente escribiendo la palabra clave COMMIT acabando en punto y coma (;).

Por otro lado se encuentra la sentencia **ROLLBACK**, que permite deshacer todos los cambios realizados en una instancia desde el último COMMIT ejecutado. Es decir, la sentencia ROLLBACK permite a un usuario volver al estado en el que se encontraba su instancia en el último COMMIT efectuado. La forma básica de ejecutar esta sentencia es mediante la palabra clave ROLLBACK seguida de punto y coma (;).

10.2.1.3.2 GRANT Y REVOKE

Estas dos sentencias están ligadas a la seguridad de los datos de una base de datos ya que permiten otorgar (GRANT) y quitar (REVOKE) diversos permisos o privilegios a los usuarios sobre los objetos de la base de datos.

Principalmente, existen dos formas de utilizar la sentencia **GRANT**, una es para otorgar privilegios de sistema a usuarios, roles y PUBLIC, mientras que la otra manera es para otorgar privilegios sobre un objeto en particular a usuarios, roles y PUBLIC. A continuación se detallan cada una de ellas.

- **Otorgar privilegios de sistema** a usuarios, roles y PUBLIC. Los privilegios de sistema son aquellos que permiten realizar ciertas operaciones en la base de datos. Algunos de ellos son: CREATE SESSION, CREATE TABLE, CREATE VIEW, CREATE TRIGGER, CREATE USER, CREATE ROLE y DROP USER. La forma básica de otorgar este tipo de privilegios es la siguiente:

```
GRANT privilegiosDeSistema/Rol/ALL PRIVILEGES
TO Usuario/Rol/PUBLIC;
```

Figura 153: GRANT - Privilegios de sistema

Como se puede observar en la *Figura 153*, se pueden otorgar privilegios de sistema, roles o todos los privilegios (*ALL PRIVILEGES*) a un usuario, rol o cualquier usuario (*PUBLIC*). A continuación se muestra un ejemplo.

```
GRANT CREATE SESSION, CREATE TABLE
TO user001;
```

Figura 154: Otorgando privilegios de sistema

En la *Figura 154*, se observa cómo se le conceden los permisos de conexión a la base de datos (*CREATE SESSION*) y creación de tablas (*CREATE TABLE*) al usuario *user001*.

- **Otorgar privilegios sobre un objeto** a usuarios, roles y PUBLIC. Los privilegios sobre un objeto son aquellos que permiten a un usuario acceder a los objetos de otro usuario. Los más importantes son los siguientes: SELECT, UPDATE, INSERT, DELETE y ALTER. La forma básica de otorgar privilegios sobre objetos es la que se muestra a continuación.

```
GRANT privilegiosSobreObjeto/ALL PRIVILEGES
ON objeto
TO Usuario/Rol/PUBLIC;
```

Figura 155: GRANT - Privilegios sobre objetos

Como se puede comprobar en la *Figura 155*, se pueden conceder distintos privilegios o todos los privilegios sobre objetos, indicando mediante ***ON*** el objeto sobre el que se otorgan

los privilegios, a un usuario, rol o cualquier usuario (PUBLIC) mediante **TO**. Seguidamente se incluye un sencillo ejemplo.

```
GRANT SELECT, UPDATE
ON ACTOR
TO PUBLIC;
```

Figura 156: Otorgando privilegios sobre objetos

En la *Figura 156* se observa cómo se conceden los privilegios *SELECT* y *UPDATE* sobre la tabla *ACTOR* a todos los usuarios.

Por último, como se había comentado anteriormente, también se pueden retirar los permisos que se otorgan con GRANT a través de la sentencia **REVOKE**. La forma básica de utilizar esta sentencia para privilegios de sistema es la que se muestra en la siguiente figura.

```
REVOKE privilegiosDeSistema/Rol/ALL PRIVILEGES
FROM Usuario/Rol/PUBLIC;
```

Figura 157: REVOKE - Retirando privilegios de sistema

Mientras que la forma básica de esta sentencia para privilegios sobre objetos es la siguiente.

```
REVOKE privilegiosSobreObjeto/Rol/ALL PRIVILEGES
ON objeto
FROM Usuario/Rol/PUBLIC;
```

Figura 158: REVOKE - Retirando privilegios sobre objetos

Seguidamente, en la *Figura 159* se incluye un ejemplo en el que se retiran los privilegios de sistema *CREATE SESSION* Y *CREATE TABLE* al usuario user001.

```
REVOKE CREATE SESSION, CREATE TABLE
FROM user001;
```

Figura 159: Retirando privilegios de sistema a user001

Para profundizar y conocer la sintaxis completa de cada una de las sentencias que se han mencionado, se incluye este **manual SQL de Oracle**[41] en el que se detalla minuciosamente el lenguaje, con numerosos ejemplos y particularidades de Oracle.

[41] http://docs.oracle.com/cd/E11882_01/server.112/e41084.pdf